Learning, Reasoning, and Planning with Neuro-Symbolic Concepts

by

Jiayuan Mao

B.Eng., Tsinghua University (2019)
S.M., Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2026

© Jiayuan Mao. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored By: Jiayuan Mao
Department of Electrical Engineering and Computer Science
September 12, 2025

Certified By: Leslie Pack Kaelbling
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified By: Joshua B. Tenenbaum
Professor of Computational Cognitive Science
Thesis Supervisor

Accepted By: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Learning, Reasoning, and Planning with Neuro-Symbolic Concepts

by

Jiayuan Mao

Submitted to the Department of Electrical Engineering and Computer Science
on September 12, 2025, in partial fulfillment of the
requirements for the degree of
DOCTOR OF PHILSOPHY

Abstract

Today's AI systems excel at pattern recognition over analog inputs such as pixels and torques, yet they lack the ability to reason deeply about abstract *concepts*, such as objects, relations, events, and actions. However, fundamental questions remain: how can we represent concepts, learn them from observations, and reason about them effectively?

This dissertation presents a concept-centric paradigm for building agents that can learn continually and reason flexibly. The agent utilizes a vocabulary of *neuro-symbolic concepts*. These concepts of objects, relations, and actions are grounded in sensory inputs and actuation outputs; they are also compositional, allowing for the creation of novel concepts through their structural combination. To facilitate learning and reasoning, the concepts are typed and represented using a combination of symbolic programs and neural network embeddings. Leveraging the complementary features of neural and symbolic representations, an agent can efficiently learn and recombine concepts to solve various tasks across different domains, ranging from 2D images, videos, 3D scenes, and robotic manipulation tasks. This concept-centric framework offers several advantages, including data efficiency, compositional generalization, continual learning, and zero-shot transfer.

This dissertation consists of three parts, where we investigate the use of such a paradigm for visual understanding, reasoning, and decision making. The entire dissertation is organized as follows. In Part I, we develop a neuro-symbolic logic reasoning framework that enables machines to learn the manipulation of concept representations grounded on raw observations, supporting applications such as logic reasoning and constraint satisfaction solving. The key innovation is a unified framework that connects deep neural networks' learning and inference with symbolic reasoning. In Part II, we introduce a neuro-symbolic approach to concept learning in visual domains, with an emphasis on vision-language tasks such as image captioning and visual question answering. Here, we explore the synergy between language and perception: how language can guide the learning of concepts and, conversely, how concepts can inform language acquisition. Finally, in Part III, building on the models introduced in Parts I and II, we extend the framework to hybrid discrete-continuous decision-making. This part focuses on new representations and planning algorithms with applications in robotic control and long-horizon reasoning.

Thesis Supervisor: Leslie Pack Kaelbling
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Joshua B. Tenenbaum
Title: Professor of Computational Cognitive Science

Contents

1 Prologue: Concepts **33**
1.1 Neuro-Symbolic Concepts . . . 35
1.2 Structure of the Dissertation . . . 37

I Foundations of Neuro-Symbolic Representation and Reasoning 41

2 Neuro-Symbolic First-Order Logic **43**
2.1 Introduction . . . 43
2.2 Neural Logic Machines (NLM) . . . 45
2.2.1 Logic Predicates as Tensors . . . 46
2.2.2 Logic Rules as Neural Operators . . . 46
2.2.3 Neural Logic Machines . . . 49
2.2.4 Expressiveness and Computational Complexity . . . 51
2.3 Experiments . . . 51
2.3.1 Baselines . . . 52
2.3.2 Family Tree Reasoning . . . 52
2.3.3 General Graph Reasoning . . . 54
2.3.4 Blocks World . . . 54
2.3.5 General Algorithms . . . 55
2.4 Sparsity and Locality in Neural Logic Machines . . . 56
2.4.1 Sparse Hypergraph Neural Networks . . . 59
2.4.2 Sparsification through Hoyer Regularization . . . 62
2.4.3 Subgraph Training . . . 62
2.4.4 Family Tree Reasoning . . . 65
2.4.5 Real-World Knowledge Graph Reasoning . . . 68
2.5 Related Work . . . 70
2.6 Discussion . . . 72

3 Neuro-Symbolic Temporal Logic **75**
3.1 Introduction . . . 75
3.2 TOQ-Nets . . . 77
3.2.1 Temporal-Relational Feature Representation . . . 79
3.2.2 Relational Reasoning Layers . . . 80
3.2.3 Temporal Reasoning Layers . . . 81
3.3 Experiments . . . 84
3.3.1 Baseline Approaches . . . 84
3.3.2 Trajectory-Based Soccer Event Detection . . . 85
3.3.3 Manipulation Concept Learning from 6D Poses . . . 91
3.3.4 Extension to Real-World Datasets . . . 93
3.4 Related Work . . . 94
3.5 Discussion . . . 95

4 Expressiveness of Neuro-Symbolic Logic Models **97**
4.1 Introduction . . . 97
4.2 Preliminaries: Planning Domain and Problem . . . 98
4.3 Goal Regression Search and Width . . . 100
4.3.1 Serialized Goal Regression Search . . . 100
4.3.2 Serialization of Goal Regression Rules . . . 102
4.3.3 Width of Search Problems . . . 103
4.4 Policy Realization . . . 104
4.4.1 Relational Neural Network Expressiveness . . . 106
4.4.2 Compilation of BWD and S-GRS . . . 108
4.4.3 Compilation of S-GRS with a Regression Rule Selector . . . 109
4.5 Problem Analysis and Results . . . 112
4.6 Related Work . . . 114
4.7 Discussion . . . 115

5 Neuro-Symbolic Continuous Constraint Satisfaction **117**
5.1 Introduction . . . 118
5.2 Compositional Diffusion Constraint Solvers . . . 120
5.2.1 Formulation and Representation of Constraint Satisfaction Problems 120
5.2.2 Compositional Diffusion Models . . . 121
5.2.3 Geometric and Physical Constraint Solving with Diffusion-CCSP . 122
5.3 Experiments . . . 124
5.3.1 2D Triangle Packing . . . 125
5.3.2 2D Shape Arrangement with Qualitative Constraints . . . 125

5.3.3 3D Object Stacking with Stability Constraints . . . 127
5.3.4 Object Packing with Robot Trajectory Constraints . . . 128
5.4 Related Work . . . 128
5.5 Discussion . . . 129

II Neuro-Symbolic Visual Concepts 131

6 Visual Concept Learning 133
6.1 Introduction . . . 133
6.2 Related Work . . . 134
6.3 Neuro-Symbolic Concept Learner . . . 137
6.3.1 Model details . . . 137
6.3.2 Training paradigm . . . 139
6.4 Experiments . . . 141
6.4.1 Visual concept learning . . . 141
6.4.2 Data-efficient and interpretable visual reasoning . . . 141
6.4.3 Generalization to new attributes and compositions . . . 143
6.4.4 Combinatorial generalization to new scenes and questions . . . 144
6.4.5 Extending to other program domain . . . 145
6.4.6 Extending to natural images and language . . . 146
6.5 Discussion . . . 147

7 Fast Visual Concept Learning with Metaconcepts 149
7.1 Introduction . . . 149
7.2 FALCON . . . 151
7.2.1 Visual Representations and Embedding Spaces . . . 153
7.2.2 Neuro-Symbolic Reasoning . . . 154
7.2.3 Learning to Learn Concepts . . . 155
7.2.4 Training and Testing Paradigm . . . 156
7.3 Experiments . . . 157
7.3.1 Dataset . . . 157
7.3.2 Baseline . . . 158
7.3.3 Results on CUB . . . 159
7.3.4 Results on CLEVR . . . 162
7.3.5 Results on GQA . . . 163
7.4 Related Work . . . 163
7.5 Discussion . . . 165

8 Concept Learning Across Domains and Modalities **167**
8.1 Introduction . . . 167
8.2 Logic-Enhanced Foundation Model (LEFT) . . . 169
8.2.1 Domain-independent LLM interpreter . . . 171
8.2.2 Domain-independent first-order logic executor . . . 172
8.2.3 Domain-specific grounding modules . . . 174
8.3 Experiments . . . 175
8.3.1 Concept learning across domains . . . 175
8.3.2 Reasoning generalization across tasks . . . 178
8.4 Related Work . . . 181
8.5 Discussion . . . 182

9 Visual Concept Lexicon **183**
9.1 Introduction . . . 184
9.2 Grammar-Based Grounded Lexicon Learning . . . 186
9.2.1 Grounded Lexicon . . . 187
9.2.2 Program Execution . . . 189
9.2.3 Joint Chart Parsing and Expected Execution (CKY-E2) . . . 189
9.2.4 Learning . . . 192
9.3 Experiment . . . 193
9.3.1 Visual Reasoning . . . 193
9.3.2 Language-driven Navigation . . . 195
9.4 Related Work . . . 198
9.5 Discussion . . . 200

III Neuro-Symbolic Action Concepts **201**

10 Neuro-Symbolic Skills **203**
10.1 Introduction . . . 204
10.2 Planning with Contacts and Mechanisms . . . 206
10.2.1 Basic Domain Representation . . . 207
10.2.2 Mechanisms . . . 208
10.2.3 Planning with Basis Operators and Mechanisms . . . 209
10.3 Learning New Mechanisms . . . 209
10.3.1 Extraction of Preconditions and Operation Sequence . . . 210
10.3.2 Sampler Learning . . . 210
10.3.3 Implementation . . . 212

10.4 Experiments . . . 213
10.4.1 Learning Mechanisms from Single Demonstrations . . . 213
10.4.2 Planning with Learned Mechanisms . . . 214
10.5 One-Shot Manipulation Strategy Learning . . . 215
10.5.1 Problem Definition . . . 218
10.5.2 Global Contact Point Matching with Pretrained Features . . . 219
10.5.3 Local Contact Point Matching with Curvature Estimation . . . 220
10.5.4 Generating Object Motions by Making Contact Analogies . . . 222
10.5.5 Experiment Setup . . . 223
10.5.6 Simulation Experiments . . . 224
10.5.7 Real-World Experiments . . . 228
10.6 Keypoint-Based Manipulation Strategy Learning . . . 229
10.6.1 Problem Formulation . . . 232
10.6.2 Keypoint Desiderata . . . 233
10.6.3 Keypoint Proposal and Verification . . . 233
10.6.4 Learning Keypoint-Centric Trajectory Prediction Models . . . 236
10.6.5 Implementation . . . 236
10.6.6 Inference-time Pipeline . . . 237
10.6.7 Simulation Experiments in Meta-World . . . 237
10.6.8 Real-World Experiments on Franka Arm . . . 239
10.7 Related Work . . . 241
10.8 Discussion . . . 244

11 Neuro-Symbolic Long-Horizon Planning 247
11.1 Introduction . . . 248
11.2 PDSketch . . . 251
11.2.1 Representation Language . . . 253
11.2.2 Model Learning and Planning with PDSketch . . . 254
11.2.3 Inducing Domain-Independent Heuristics . . . 256
11.3 Experiments . . . 257
11.3.1 BabyAI . . . 258
11.3.2 Robot Painting . . . 259
11.4 Behavior from Language and Demonstration . . . 261
11.4.1 Problem Formulation . . . 263
11.4.2 Behavior Description Learning . . . 264
11.4.3 Classifier and Policy Learning . . . 266
11.4.4 Planning and Execution . . . 267
11.4.5 Simulation Experiments . . . 267

11.4.6 Real-World Experiments . . . 270
11.5 Related Work . . . 271
11.6 Discussion . . . 274

12 Hybrid Imperative-Declarative Action Representations 275
12.1 Introduction . . . 276
12.2 The Crow Definition Language (CDL) . . . 279
12.2.1 Imperative Behavior Scripting in CDL . . . 281
12.2.2 Nondeterminsitic Behaviors and Soundness Checking . . . 282
12.2.3 Subgoal Ordering and Promotion . . . 284
12.2.4 Variable Binding . . . 287
12.2.5 Commitment . . . 288
12.3 The Crow Interpreter . . . 289
12.3.1 Problem Hardness and Crow Runtime . . . 291
12.3.2 Experiments in NAMO . . . 292
12.4 Related Work . . . 293
12.5 Discussion . . . 294

13 Epilogue: All You Need is Everything 295

14 Encore: Everything is One Thing 299

A Appendix for Neural Logic Machines 337
A.1 Training Method and Curriculum Learning . . . 337
A.1.1 Training method . . . 337
A.1.2 Curriculum learning guided by exams and fails . . . 338
A.2 Implementation details and hyper-parameters . . . 340
A.2.1 Residual connection. . . . 340
A.2.2 Hyper-parameters for NLM . . . 340
A.2.3 Hyper-parameters for MemNN . . . 340
A.2.4 Data generation . . . 341
A.2.5 Blocks World . . . 342
A.2.6 Accuracy discussion . . . 342
A.3 Neural Logic Machines (NLM) extensions . . . 342
A.4 Realization of Horn Clause . . . 343
A.5 Sample Blocks World Rules . . . 345
A.6 Implement NLM in TensorFlow . . . 346

B Appendix for Sparse and Local Neural Logic Machines **349**
B.1 Experimental configuration . . . 349
B.2 SpaLoc's results with standard errors . . . 350
B.3 Implementation of subgraph samplers . . . 350
B.4 Ablation study on sparsification loss . . . 351
B.5 Ablation study on SpaLoc's maximum arity . . . 351
B.6 Ablation Study on Label Adjustment . . . 352
B.7 Calculation of the information sufficiency . . . 352
B.8 Definition of Relations in Family Tree . . . 353

C Appendix for Temporal and Object Quantification Networks **355**
C.1 Implementation Details . . . 355
C.1.1 TOQ-Nets . . . 355
C.1.2 STGCN . . . 356
C.1.3 STGCN-MAX . . . 356
C.1.4 STGCN-2D . . . 357
C.1.5 STGCN-LSTM . . . 357
C.1.6 Space-Time Region Graph Networks . . . 358
C.1.7 Non-Local Neural Networks . . . 358

D Appendix for Expressivity of Neuro-Symbolic Logic Models **361**
D.1 Proofs and Discussions for Search Complexity . . . 361
D.1.1 Complete and Optimal Version of S-GRS . . . 361
D.1.2 Proof for Theorem 4.3.1: S-GRS Optimality and Completeness . . . 362
D.1.3 Making Non-SOS Problem Serializable . . . 363
D.1.4 Proof for Theorem 4.3.2: Polynomial Hardness for SOS-k problems. 363
D.1.5 Additional Discussion . . . 364
D.1.6 Connection with Width-Based Forward Search IW(k) . . . 365
D.2 Proofs and Discussions for Policy Circuit Complexity . . . 367
D.2.1 Proof of Theorem 4.4.1: Compilation of BWD . . . 367
D.2.2 Proof of Theorem 4.4.2: Compilation of S-GRS . . . 368
D.2.3 More Discussion . . . 368
D.3 Analysis of AI Planning Problems . . . 370
D.3.1 SOS Width Analysis . . . 372
D.4 Experimental Details . . . 373
D.4.1 Training details . . . 373

E Appendix for Diffusion Continuous Constraint Satisfaction Solver 375
E.1 Problem Domains and Data Generation . . . 375
E.1.1 2D Triangle Packing . . . 375
E.1.2 2D Shape Arrangement with Qualitative Constraints . . . 376
E.1.3 3D Object Stacking with Stability Constraints . . . 377
E.1.4 3D Robot Packing with Robots . . . 378
E.1.5 Discussion: Sample Complexity of Diffusion-CCSP . . . 378
E.1.6 Discussion: Partial Observability . . . 379
E.2 Implementation and Training Details . . . 379
E.2.1 Discussion: Individual Constraint Diffusion Training and Finetuning 380
E.2.2 Discussion: Weights for Energy Function Composition . . . 380
E.2.3 Discussion: Local Optima in Optimization . . . 380
E.3 Failure Case Analysis . . . 381
E.4 Additional Sampler Comparison . . . 381
E.5 Integration with Task and Motion Planning Algorithms . . . 382
E.6 Learned Gradient Fields . . . 385

F Appendix for Visual Concept Learning 393
F.1 CLEVR Domain-Specific Language . . . 393
F.2 Semantic Parsing . . . 393
F.3 Program Execution . . . 396
F.4 Optimization of the Semantic Parser . . . 397
F.5 Curriculum Learning Setup . . . 398
F.6 Ablation Study . . . 399
F.6.1 Semantic parsing accuracy. . . . 399
F.6.2 Impacts of the ImageNet pre-training. . . . 399
F.6.3 Data Efficiency and Object-based Representations . . . 400
F.7 Extending to Other Scene and Language Domains . . . 401
F.7.1 Minecraft Dataset . . . 401
F.7.2 VQS Dataset . . . 402
F.8 Visualization of execution traces and visual concepts . . . 403

G Appendix for Fast Visual Concept Learning with Metaconcepts 413
G.1 Model Details . . . 413
G.1.1 Remarks on the overall model structure . . . 413
G.1.2 Concept Embedding . . . 414
G.1.3 Prior Distribution . . . 414
G.1.4 FALCON-G . . . 414

G.1.5 FALCON-R . . . 416
G.1.6 Semantic parser . . . 417
G.1.7 Neuro-Symbolic Progrm Execution . . . 419
G.1.8 Baseline Implementation . . . 419
G.2 Training Details . . . 420
G.3 Experiment Details . . . 421
G.3.1 The CUB Dataset . . . 421
G.3.2 The CLEVR Dataset. . . . 424
G.3.3 The GQA dataset . . . 427
G.3.4 The connections between the box embedding and our formulation . 428
G.4 Ablation Details . . . 431
G.4.1 The Effect of the Number of Base Concepts . . . 431
G.4.2 The Effect of The Number of Related Concepts in Supplemental Sentences . . . 432
G.5 Failure Mode Details . . . 433
G.5.1 Failure mode in detection module . . . 433
G.5.2 Failure mode in semantic parser . . . 434
G.5.3 Failure mode in concept learning module . . . 434

H Appendix for Concept Learning Across Domains and Modalities 439
H.1 LEFT . . . 439
H.1.1 Function definitions . . . 439
H.1.2 Broader impact . . . 439
H.2 Experiments . . . 441
H.2.1 Error bars . . . 441
H.2.2 Transfer task construction . . . 441
H.3 Model . . . 442
H.3.1 Implementation details . . . 442
H.3.2 Training details . . . 442

I Appendix for Visual Lexicon Learning 443
I.1 Domain Specific Languages and Neuro-Symbolic Reasoning . . . 443
I.1.1 Visual Reasoning DSL . . . 443
I.1.2 Language-Driven Navigation DSL . . . 449
I.2 Delve Into Expected Execution . . . 452
I.2.1 CKY-E2 In An Arithmetic Domain . . . 452
I.2.2 Formal Properties of CKY-E2 . . . 453
I.2.3 Connection with Other Parsing Models . . . 455

I.3 Experimental Setup And Analysis . . . 456
I.3.1 Visual Reasoning: CLEVR . . . 456
I.3.2 Language-Driven Navigation: SCAN . . . 458

J Appendix for Manipulation Mechanisms 461
J.1 Method Detail . . . 461
J.1.1 Planning Algorithm . . . 461
J.1.2 Briefly Dynamic Manipulation . . . 462
J.1.3 Basis Operations . . . 462
J.1.4 Samplers for Generic Operations . . . 463
J.2 Experiment Detail . . . 465
J.2.1 Mechanism Learning Setup . . . 465
J.2.2 Sampler Learning . . . 465
J.2.3 Physical Robot Deployment . . . 467

K Appendix for PDSketch 469
K.1 Additional BabyAI Results . . . 469
K.1.1 Data Efficiency and Variance Analysis . . . 469
K.1.2 On-Policy Learning . . . 470
K.2 PDSketch Language . . . 471
K.2.1 Predicate and Action Definition . . . 471
K.2.2 Expressions . . . 481
K.3 Domain-Independent Heuristic . . . 484
K.4 Experimental Setup . . . 488
K.4.1 BabyAI . . . 488
K.4.2 Painting Factory . . . 490

L Appendix for Behavior from Language and Demonstration 493
L.1 BLADE Details . . . 493
L.1.1 Behavior Description Generation with LLMs . . . 493
L.1.2 Predicate Generation with LLMs . . . 494
L.1.3 Temporal Segmentation . . . 494
L.1.4 Abstract Verification . . . 497
L.1.5 Automatic Predicate Annotation . . . 498
L.1.6 Classifier Implementation . . . 498
L.1.7 Policy Implementation . . . 500
L.1.8 Planner Implementation . . . 501
L.2 Simulation Experiment Details . . . 502

L.2.1 Task Design . . . 502
L.2.2 Baseline Implementation . . . 504
L.3 Qualitative Examples . . . 505
L.4 Real World Experiment Details . . . 505
L.4.1 Task Design . . . 505
L.4.2 Qualitative Examples of Novel States . . . 509
L.5 Prompts for Baselines . . . 509

M Appendix for Hybrid Imperative-Declarative Action Representations 529
M.1 Refinements and The Crow Interpreter . . . 529
M.1.1 The Completeness and Soundness of Crow . . . 532
M.2 Formulations and Algorithms for Constraint Satisfaction Problems . . . 533
M.2.1 Sampling-based Continuous Constraint Satisfaction Problem Solver 533
M.3 Proofs for Planner Complexity . . . 534
M.4 Implementation Details for NAMO . . . 536
M.5 Manual of CDL . . . 539
M.5.1 Key Concepts . . . 539
M.5.2 State Representation: Types and Features . . . 540
M.5.3 Functions . . . 542
M.5.4 Non-Data-Oblivious Functions and Generators . . . 543
M.5.5 Behavior Effects . . . 544
M.5.6 Examples . . . 546

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

1-1 Three types of neuro-symbolic concepts. 34
1-2 The neuro-symbolic concept-centric framework. 36

2-1 Illustration and predicates of the blocks world. 45
2-2 An illustration of Neural Logic Machines (NLMs). 47
2-3 An illustration of the computational block inside NLM for binary predicates. 50
2-4 The overall training pipeline of SpaLoc. 58
2-5 A running example of a single-layer SpaLoc. 60
2-6 Subgraph sampling and label calibration in SpaLoc. 63
2-7 Memory usage and inference time of SpaLoc. 66

3-1 Describing complex events with FO-LTL formula. 76
3-2 Overview of TOQ-Nets. 78
3-3 Illustration of relational reasoning layers and temporal reasoning layers. . . 80
3-4 Generalization to soccer environments with a different court size and agent speeds. 88
3-5 Comparing # of reasoning layers in TOQ-Nets. 90
3-6 Relevant features in temporal layers in TOQ-Nets. 91
3-7 Comparing different models with different time stretching factors on the RLBench dataset. 93

4-1 Illustration of the Blocks World domain and search trees. 99
4-2 The input and output of a relational neural network (*RelNN*) policy. 105
4-3 State-dependent regression rule selector in the Blocks World domain. . . . 109

5-1 Solving Continuous Constraint Satisfaction Problems by Composing Diffusion Models. 119
5-2 Continuous constraint satisfaction problem (CCSP) in robot planning. . . . 120
5-3 Illustration of the denoising neural network in Diffusion-CCSP. 123
5-4 Illustration of four domains. 125
5-5 Quantitative comparisons of constraint solvers. 126

5-6 Visualization of learned energy functions of selected geometric and qualitative constraints. 127

6-1 Curriculum learning of visual concepts. 135
6-2 Neuro-symbolic reasoning. 137
6-3 Grounding of visual concepts. 138
6-4 Curriculum learning and illustrative execution of NS-CL. 140
6-5 Results of compositional generalization on CLEVR. 143
6-6 Compositional generalization examples on CLEVR. 144
6-7 Results on VQS. 146
6-8 Concepts learned from VQS. 146

7-1 Three illustrative examples of our fast concept learning task. 150
7-2 Overview of FALCON-G. 152
7-3 Different models for predicting concept embedding e_c in FALCON. 158
7-4 Qualitative results of FALCON-G on CUB and CLEVR. 161

8-1 Overview of LEFT. 168
8-2 Domain-independent and domain-specific modules in LEFT. 170
8-3 Evaluation tasks of LEFT. 175
8-4 Illustration of three zero-shot generalization tasks. 180

9-1 Illustration of G2L2. 185
9-2 Parsing and program execution of G2L2. 186
9-3 Grounded lexicon in G2L2. 187
9-4 Illustration of expected execution of two semantic programs. 190

10-1 Reusable manipulation strategies. 205
10-2 Planning with basis manipulation operations and mechanisms. 206
10-3 Syntax for basis operations and mechanisms. 207
10-4 Learning of manipulation strategies. 209
10-5 Illustration of six mechanisms learned by our algorithm. 214
10-6 Illustration of compositional tasks. 215
10-7 Overview of MAGIC. 216
10-8 The overall pipeline of MAGIC. 217
10-9 Global and local contact point matching. 219
10-10 Observation scale for curvature computation. 221
10-11 Qualitative examples of contact point correspondence matching. 224
10-12 Real-world experiment setup. 228

10-13Average success rates of real-world experiments. 228
10-14Contact point matching results on real-world objects. 229
10-15Keypoint abstraction using large models for object-relative imitation learning.230
10-16Overview of KALM. 231
10-17Testing tasks in Meta-World. 238
10-18Data efficiency results on Meta-World. 239
10-19Testing tasks in the real world. 240

11-1 The life cycle of a PDSketch model. 248
11-2 Structures in transition models. 249
11-3 Factorized state representations and transition models. 250
11-4 A minimal example of defining derived features and predicates with blanks. 253
11-5 A computation graph of a partial definition of the action "move-into." . . . 255
11-6 A screenshot of the BabyAI environment. 257
11-7 Data efficiency and planning efficiency of PDSketch. 257
11-8 The PDSketch definition of the "forward" action and the corresponding computation graph for the PDS-Abs model. 258
11-9 Encoding of the painting factory domain. 260
11-10Generalization results on the painting factory domain. 260
11-11Behavior from language and demonstration. 262
11-12Overview of BLADE. 263
11-13Behavior descriptions learning in BLADE. 265
11-14Generalization tasks in CALVIN. 268
11-15Domains and results on real-world evaluations. 271
11-16Real world planning and execution. 272

12-1 NAMO problems and solutions. 276
12-2 Illustration of the refinement operator. 286
12-3 Results on NAMO problems. 292

D-1 Illustration of the branching factor caused by tracking multiple resulting states after achieving a subgoal. 363
D-2 Illustration of the compilation of backward search into a *RelNN* policy. . . . 367

E-2 Example rearrangements of rectangles with 11 types of qualitative constraints.376
E-3 Example stable configurations of rectangles under three constraints. 377
E-4 Example configuration of 3D shapes under the collision-free constraints. . . 378
E-9 Number of Diffusion-CCSP runs it takes to solve 100 CCSPs. 382
E-1 Example collision-free configurations of triangles. 383

E-5 Failure and success samples for Task 2: 2D shape rearrangement with qualitative constraints. . . . 386
E-6 Analysis of how the failure rate of qualitative constraints correlates with the area of its defining region. . . . 387
E-7 Failure and success samples for Task 1: 2D triangle packing. . . . 388
E-8 Failure and success samples for Task 3: 3D object stacking with stability constraints. . . . 389
E-10 Illustration of a simple task and motion planning problem. . . . 390
E-11 Example of a partially specified plan skeleton in task and motion planning, together with the corresponding set of constraints. . . . 390
E-12 Visualization of energy and gradient functions of trained diffusion models for qualitative constraints (Part 1). . . . 391
E-13 Visualization of energy and gradient functions of trained diffusion models for qualitative constraints (Part 2). . . . 392

F-1 Examples from the Minecraft dataset. . . . 401
F-2 Examples from the VQS dataset. . . . 403
F-3 Visualization of the execution trace generated by NS-CL on the CLEVR dataset. . . . 408
F-4 Visualization of the execution trace generated by NS-CL on the Minecraft VQA dataset. . . . 409
F-5 Visualization of the execution trace generated by NS-CL on the VQS dataset. 410
F-6 Concepts learned on the CLEVR dataset. . . . 411
F-7 Concepts learned on the Minecraft dataset. . . . 412

G-1 An overview of the overall training paradigm of FALCON. . . . 422
G-2 A visualization of fast concept learning from biased data with supplemental sentences on CLEVR. . . . 426
G-3 Two implementations of graph $G_{concept}$. . . . 427
G-4 Fast concept learning performance under different percentages of related concepts in the supplemental sentence. . . . 433
G-5 A visualization of three failure cases of the detection module. . . . 435
G-6 A visualization of two failure cases of the concept learning module. . . . 436

J-1 Illustration of three briefly-dynamic manipulation scenarios in the paper. . . . 462
J-2 The modeling of the robot basis operation *move-with-contact* using a STRIPS-like syntax. . . . 463
J-3 Breakdown of samples produced by different samplers for the hook-using task. 466

J-4 Our real-world setup. 467

K-1 A graphical illustration of the hFF heuristics. 485

L-1 Our real-robot setup. 506
L-2 Training and testing states for the boil-water domain. 507
L-3 Training and testing states for the make-tea domain. 508
L-4 BLADE and baseline performance on an abstract goal generalization task in the CALVIN environment. 510
L-5 BLADE and baseline performance on the partial observability generalization task in the CALVIN environment. 511
L-6 BLADE and baseline performance on the geometric constraint generalization task in the CALVIN environment. 512

M-1 Example planning trace of CROW on the Sussman anomaly problem. . . . 550

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

2.1 Results on family tree and graph reasoning. 53
2.2 Results on blocks world and algorithmic reasoning tasks. 55
2.3 Results (per-class accuracy) on family tree reasoning benchmarks. 65
2.4 Performance and runtime complexity of SpaLoc. 66
2.5 Comparison of different subgraph samplers in SpaLoc. 66
2.6 Results (AUC-PR) on real-world knowledge graph inductive reasoning datasets from GraIL. 68
2.7 Results (AUC-PR) on transductive link prediction on real-world knowledge graphs. 69

3.1 A running example of different temporal quantification formulas that TOQ-Nets can realize. 83
3.2 Results on the soccer event dataset. 85
3.3 Generalization results on the soccer event dataset. 86
3.4 Results on the soccer event dataset for baselines with different capacities . . 89
3.5 Few-shot learning on the RLBench dataset. 92

4.1 Width and circuit complexity analysis for 6 problems widely used in AI planning communities. 112
4.2 Success rate of learned policies in different environments. 112

5.1 Average number of calls to Diffusion-CCSP in order to solve the full TAMP problem. 128

6.1 Visual reasoning frameworks on the CLEVR dataset. 136
6.2 Results of visual concept grounding on CLEVR. 142
6.3 Data-efficiency comparison of different models on CLEVR. 142
6.4 Results on CLEVR. 143
6.5 Results on CLEVR object retrieval. 145

7.1 Fast concept learning performance on CUB. 160

7.2 Fast concept learning with only visual examples on CUB. 160
7.3 Fast concept learning performance on CLEVR. 163

8.1 LEFT's first-order logic language. 172
8.2 Results comparing LEFT to prior top-performing neuro-symbolic and end-to-end works. 177
8.3 Results on natural language tasks. 177
8.4 Results on SR3D in data efficient settings. 178
8.5 Results on CLEVR in data efficient settings. 179
8.6 Zero-shot generalization of LEFT to three unseen tasks. 180

9.1 Results on CLEVR. 193
9.2 Results on SCAN. 196

10.1 List of basis operations. 208
10.2 Planning time comparison for manipulation strategies. 213
10.3 Average success rate on novel compositional tasks in 10 runs. Timeout is 10 minutes. 215
10.4 Average success rates (%) on the scooping, hanging, and hooking tasks. . . 224
10.5 Average success rates (%) of the hanging task (intra-category transfer vs. inter-category transfer). 225
10.6 Ablation of the choice of pretrained image features. 226
10.7 Comparison between MAGIC and KETO. 227
10.8 Ablation studies of different modules in MAGIC. 227
10.9 Few-shot learning results in Meta-World. 237
10.10 Results on real-world experiments. 240

11.1 The planning success rate of different models on BabyAI. 257
11.2 Model performance on the painting factory domain. 260
11.3 Generalization results on CALVIN. 268
11.4 Ablation on state classifier learning on CALVIN. 270

A.1 Hyper-parameters for reinforcement learning tasks. 338
A.2 Hyper-parameters for Neural Logic Machines. 341

B.1 Hyper-parameters for SpaLoc. 349
B.2 SpaLoc's results with standard errors on real-world knowledge graph inductive reasoning datasets from GraIL (Table 2.6). 350
B.3 SpaLoc's results with standard errors of transductive link prediction on real-world knowledge graphs (Table 2.7). 350

B.4 Comparison of different sparsification losses. 351
B.5 Comparison (Per-class Accuracy) of SpaLoc with different max arities on family tree reasoning benchmarks. 351
B.6 Comparison (AUC-PR) of SpaLoc with different max arities on real-world knowledge graph inductive relation prediction task. 352
B.7 Comparison (per-class accuracy) for different label calibration methods. . . 352

C.1 Ablation study of the STGCN-2D model on the soccer event dataset 357
C.2 Ablation study of the STGCN-2D model on the soccer event dataset 357
C.3 Ablation study of different kernel sizes for the STGCN-LSTM model on the RLBench dataset 358
C.4 The STGCN architecture used in the chapter. 359
C.5 The Space-Time Region Graph architecture used in the chapter. 360
C.6 The Non-local Neural Networks architecture used in the chapter. 360

D.1 Width and circuit complexity analysis for 6 problems widely used in AI planning communities. 370

E.1 Average runtime (seconds) of models in a batch of 100 CCSPs. 383

F.1 All operations in the domain-specific language for CLEVR VQA. 405
F.2 The type system of the domain-specific language for CLEVR VQA. 406
F.3 A step-by-step running example of the recursive parsing procedure. 406
F.4 All operations in the domain-specific language for CLEVR VQA. 407
F.5 Results on the Minecraft VQA dataset. 407

G.1 All program operations supported in CLEVR in FALCON. 417
G.2 Example questions and the programs recovered by the semantic parser in FALCON. 418
G.3 Templates for generating descriptive sentences, supplemental sentences and test questions in CUB. 423
G.4 The continual concept learning performance on the CUB dataset. 424
G.5 The continual concept learning performance with only visual examples on the CUB dataset. 424
G.6 Templates for generating descriptive sentences, supplemental sentences and test questions in CLEVR. 425
G.7 Ablated study of FALCON tasks w/. and w/o. supplemental sentences on the biased CLEVR dataset. 426

G.8 Templates for generating descriptive sentences, supplemental sentences and test questions in GQA. . . . 428
G.9 Fast concept learning performance on the GQA dataset. . . . 429
G.10 Fast concept learning with only visual examples, evaluated on the GQA dataset. . . . 429
G.11 AUC-ROC score of classifying entailment relationship based on concept embeddings from different embedding spaces. . . . 431
G.12 Fast concept learning (with supplemental sentence) performance under different number of base concepts. . . . 432
G.13 Fast concept learning performance under different percentage of related concepts in the supplemental sentence. . . . 432
G.14 A visualization of three failure cases of the semantic parser. . . . 437

H.1 Function definitions for first-order logic programs in LEFT. . . . 440
H.2 Results on the CLEVR dataset with standard errors. . . . 441

I.1 The type system of the domain-specific language for visual reasoning. . . . 444
I.2 All operations in the domain-specific language for visual reasoning. . . . 445
I.3 An illustrative execution trace of a program by G2L2 program executor. . . . 446
I.4 The learned lexicon entries associated with each word for a simple sentence. 449
I.5 All operations in the domain-specific language for language-driven navigation. 449
I.6 Sample semantic programs generated by the enumeration process based on our language-driven navigation DSL. . . . 451
I.7 A set of candidate lexicon entries and their weights in a simple arithmetic domain. . . . 452
I.8 Four candidate derivations of a simple sentence. . . . 452

J.1 Success rate of our physical robot setup on two representative mechanisms. 467

K.1 The planning success rate for different models using different amount of training data. . . . 470

L.1 Comparison of manually defined and automatically generated predicates for CALVIN. . . . 495
L.2 Comparison of manually defined and automatically generated predicates for the boil-water domain. . . . 495
L.3 Comparison of manually defined and automatically generated predicates for the make-tea domain. . . . 496

Chapter 1

Prologue: Concepts

One of the long-term goals of artificial intelligence (AI) is to build machines that can continually learn new knowledge from their experiences, ground the knowledge in the physical world, and apply the knowledge to their reasoning across different tasks, modalities, and environments. The desired capability of such agents includes, but is not limited to, describing aspects of the perceived world state, answering queries about the state of the world, making plans to achieve certain goals, and executing plans in the physical world. We want such machines to be able to learn and solve a wide variety of tasks in different environments, leveraging a feasible amount of data from multiple modalities.

In recent years, we have seen great success in neural network-based "end-to-end" learning methods, but most of them are tailored to particular tasks and environments, for example, segmenting and categorizing images, answering natural language queries about documents, and playing video games and board games. These systems are usually founded on the idea of *intelligence as fitting models to datasets*, relying on relatively simple and monolithic training and inference algorithms (e.g., a single neural network trained by stochastic gradient descent). As a result, their success in domain-specific applications relies on the availability of large-scale datasets on the particular task of interest. However, annotating high-quality data for reasoning, planning, and control in visual and physical domains is usually labour-intensive and, in some cases, infeasibly costly. Thus, there has been limited success in extending these methods to building embodied generalist agents in a wide variety of domains.

In contrast, this dissertation presents an alternative idea:

intelligence is founded on world model learning and performance-time reasoning.

We draw on rich traditions from philosophy and cognitive science that identify *concepts* as the basic building blocks of such world models and the reasoning process (for example, see readings from Margolis and Laurence [1999]). For humans, we acquire concepts from the

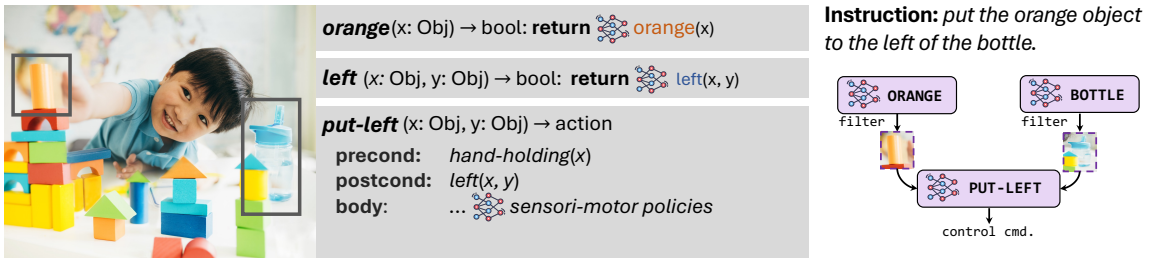


Figure 1-1: **Three types of neuro-symbolic concepts.** (Left) Illustration of three types of neuro-symbolic concepts: object properties (e.g., "*orange*"), relations (e.g., "*left*"), and actions (e.g. "*put-left*"). (Right) An *end-to-end differentiable* computation graph that composes concepts.

interaction of our evolved cognitive architecture and built-in inductive biases with our own experiences in the world, including both our direct percepts and what we learn socially and culturally from interacting and communicating with other humans. Our minds then compose these basic units to form sophisticated compound thoughts: beliefs, desires, and plans.

Importantly,

concepts form an abstract and compositional model of the world.

Abstraction and compositionality together enable systematic generalization for learning and reasoning: abstraction allows knowledge to extend beyond particular instances, so that, for example, in object-based abstraction, physical rules learned for a specific object (e.g., Apple#0510) can be applied to entirely new objects. Compositional abstractions allow individual pieces of knowledge to be applied to new situations and combined to form new knowledge, enabling a compact (and therefore efficiently learnable) description of an environment's states and transition rules. For example, states can be factorized into multiple dimensions, such as dynamics, temperature, and electrical charge, with each concept naming and describing a component of the situation. Through performance-time reasoning, generalization over situations arises from composing knowledge across different aspects of the situations.

In this dissertation, we will focus on learning a system of useful concepts for reasoning from two primary sources of knowledge: physical knowledge and linguistic knowledge. In particular, we build physical concepts on fundamental principles of physical abstractions such as objects, events, and actions. We also consider linguistic concepts at the level of individual word meanings (e.g., nouns, verbs, etc.), and treat their combinations similarly to the combination of words into phrases and sentences. We will apply this framework to visual reasoning in 2D, 3D, motion, and video data, as well as in diverse decision-making tasks spanning virtual agents and real-world robotic manipulation.

1.1 Neuro-Symbolic Concepts

Our technical proposal in this dissertation is to build a *neuro-symbolic concept* representation and leverage that for learning and reasoning. Technically, each concept is a discrete symbol (e.g., a physical property or a natural language phrase) that can be grounded onto subsets of the external physical world. For example, an object concept "orange" grounds to the sets of orange objects; an object relation concept "left" grounds to all object pairs (A, B) such that A is left of B; an action concept "put-left" grounds to all agent action sequences that move the object currently being held to a position on the left of the reference object. This includes pushing the object to the goal, holding it forcefully using tools, and so forth. As illustrated in Fig. 1-1, during performance-time reasoning, our *concept-centric* framework operates at a higher level of abstraction with this vocabulary of *neuro-symbolic concepts*.

In order to systematically represent the grounding of concepts and how they can be composed, we formally represent each concept c as a tuple of

$$c = \langle \textit{parameter}, \textit{program}, \textit{neural-nets} \rangle.$$

In essence, our concept representation integrates neural network representations (for instance, vector embeddings), which ground concepts in visual and physical representations, and symbolic representations (particularly, parameterized programs), which characterize how various concepts can be combined. Shown in Fig. 1-2a, the concept "orange" is represented as

$$\textit{orange} = \langle \{x\}, \textit{filter}(x, \textbf{ORANGE}), \{\textbf{ORANGE}\} \rangle,$$

where **ORANGE** denotes some neural network weights that can be used by the built-in *filter* function to classify orange objects. In practice, for example (Chapter 6), this *filter* function can be implemented by computing the cosine similarity between the neural representation of object x and a vector embedding of **ORANGE**.

Similarly, a relational concept (e.g., a prepositional phrase) "left-of" can be defined as

$$\textit{left-of} = \langle \{x, y\}, \textit{relate}(x, y, \textbf{LEFT-OF}), \{\textbf{LEFT-OF}\} \rangle.$$

The concept relates two objects x and y. The *relate* function will compute the cosine similarity between **LEFT-OF** and a pairwise representation between (x, y) to determine whether x is left of y in the scene. Actions (e.g., verbs) would have three parts in their programs: a controller that can generate sequences of robot control commands, and the pre- and post-conditions for the action. For example, an action "put-left-of" can be represented



orange = $\langle\{x\}, \text{filter}(x, \text{ORANGE}), \{\text{ORANGE}\}\rangle$

Instruction: *put the orange object left of the bottle.*

(a) Visual grounding of concepts.

(b) Composition of learned object and action concepts.

Figure 1-2: **The neuro-symbolic concept-centric framework.** In a neuro-symbolic concept-centric framework, different concepts such as object categories, properties, relations, and actions are represented as a combination of programmatic and neural representations. In (a), the neural representation connects the concept with sensory and actuation representations. In (b), different concepts can be combined to form new compound concepts.

as:

$$\textit{put-left-of} = \langle\{x, y\}, \{\textit{pre} = \textit{holding}(x), \textit{post} = \textit{left-of}(x, y), \\ \textit{body} = \textit{control}(\textbf{PUT-LEFT-OF})\}, \{\textbf{PUT-LEFT-OF}\}\rangle.$$

Here, the preconditions and post-conditions of the action can be described with formulas composed from other object-level and relational concepts: *holding* and *left-of*.

Illustrated in Fig. 1-2b, this representation of concepts enables us to combine existing concepts adhering to symbolic functional composition rules as performance time for reasoning, to form compound concepts such as $\textit{orange}(x)$ **and** $\textit{cylinder}(x)$ (orange cylinders), or more complex ones such as $\textit{orange}(x)$ **and** $\textit{bottle}(y)$ **and** $\textit{put-left-of}(x, y)$ (put the orange object left of the bottle). To support the formal compositionality of different concepts, all parameters and outputs are typed with primitive types (including objects, events, actions, Booleans, and integers). For instance, object concepts are represented as functions that take the perceptual representation of an object as input and predict classification scores as output, indicating whether the object has the concept. Relational concepts are associated with classifiers that classify object pairs. Meanwhile, action concepts are linked with preconditions (circumstances under which the action can be executed), postconditions (the outcomes of executing the action), and controllers that generate agent actions based on the current perceptual state. Depending on the domain and the task, one can choose to implement different primitive operations (e.g., *filter* in a visual recognition context).

1.2 Structure of the Dissertation

This general neuro-symbolic concept framework enables us to simultaneously better *understand intelligence* as a phenomenon and *build intelligence* with machine learning and reasoning.

First, the neuro-symbolic concept framework allows us to better understand the principles of representation and computation for reasoning by examining the roles of *abstraction* and *compositionality*. For example, some of my theoretical work has shown that by introducing spatial and temporal factorizations that break down scenes into entities and trajectories into subgoals, we can obtain provable polynomial circuit complexity bounds for environmental transition-rule learning [Dong *et al.*, 2019; Luo *et al.*, 2022a; Yau *et al.*, 2025, see Chapter 2], policy learning [Mao *et al.*, 2023c, see Chapter 4], and planning [Mao *et al.*, 2024, see Chapter 12].

Second, the neuro-symbolic concept framework suggests new paradigms for building agents that can adapt to new concepts (words, objects, skills) quickly, with on the order of 1 to 10 examples [Mei *et al.*, 2022, see Chapter 7], [Liu *et al.*, 2025, see Chapter 10], and generalize rapidly to novel states, goals [Mao *et al.*, 2022; Liu *et al.*, 2024b, see Chapter 11], and even embodiments [Ko *et al.*, 2024]. Compared with purely end-to-end learning methods, the *abstraction* and *compositionality* of neuro-symbolic concepts make them much better suited for generalist agent learning. They simultaneously supports four properties required by generalist agents: data efficiency (since collecting labeled data is especially difficult), compositional generalization (since the number of possible scenes and tasks can be exponentially large as the number of objects and their properties increases), continual learning (since the system must be able to learn and adapt gradually), and transfer learning (since we want systems that can leverage knowledge across different tasks).

Part I Part I presents the foundations of neuro-symbolic learning and reasoning. It contains foundational ideas for building neuro-symbolic models and inference algorithms that enable machines to learn and manipulate concept representations grounded in raw observations. This part emphasizes unifying symbolic reasoning (logic and constraint satisfaction) with neural network learning and inference within a single framework.

In particular, in Chapter 2, we introduce neural logic machines (NLMs), a novel neural network architecture that can be trained to realize first-order logic rules and generalize from small-scale tasks (such as sorting short arrays) to large-scale tasks (such as sorting longer arrays). This framework establishes a deep connection between first-order quantification structures in logic and neural network architectures; it also motivates the development of important techniques for differentiable program execution that will appear in later chapters,

such as Chapter 6 and Chapter 11.

In Chapter 3, we extend the NLM architecture to *linear temporal logic* structures, modeling important temporal relations among events. The key innovation in this chapter is a differentiable realization of temporal operators such as "next" and "eventually." This extension is important because it paves the way for modeling temporal structures of actions, which we will introduce in Part III.

In Chapter 4, we examine the theoretical aspects of these neuro-symbolic logic representations, focusing on the question: "What functions can a neural logic machine realize, and what planning problems can a neural logic machine solve?" We will begin with a theoretical expressiveness equivalence between NLMs and first-order logic. Next, by introducing a new concept called "subgoal serializability," we show under what circumstances the optimal policy for a planning problem may exist as a small feed-forward NLM circuit. This provides a theoretical answer to the limitations of NLMs and sheds new light on the trade-offs between world modeling (declarative planning) and policy learning (imperative procedures) in decision-making. These insights further motivate the development of hybrid imperative–declarative representations of actions in Chapter 12.

Finally, in Chapter 5, we examine deep generative models (in particular, diffusion models) and connect them to generators in constraint satisfaction solving. This connection enables us to leverage advances in generative model training and inference to build novel tools that can learn to solve continuous-domain constraint satisfaction problems — key problem classes that will arise in robotic planning in Part III.

All of the methods developed in this part make minimal assumptions about input representations, beyond the notion of entities (objects) and time. For example, each object can be represented either with human-interpretable state features (e.g., poses and colors) or as an image patch, as inputs to a neural logic machine or a diffusion-based constraint satisfaction solver. Concepts such as "subevents" are thus *implicitly* represented in neural network weights and *implicitly* manipulated through feed-forward computation. In the following chapters, we will introduce natural language as an additional modality to facilitate the learning and reasoning of these concepts.

Part II Part II presents a neuro-symbolic concept learning framework for visual domains, with a particular focus on vision–language tasks such as image captioning and visual question answering. This part explores the bidirectional synergy between language and visual concept learning: how natural language provides a powerful scaffold for guiding the acquisition of visual concepts, and how structured concept representations can in turn support the acquisition of language.

In particular, in Chapter 6, we introduce the neuro-symbolic concept learner (NS-

CL), a framework that learns visual concepts, words, and semantic parsing of sentences without explicit supervision on any of them. Instead, NS-CL learns by simply looking at images and reading paired questions and answers. NS-CL builds on ideas from Chapter 2, particularly the differentiable execution of programs resembling the reasoning process of natural language queries. Compared to alternative approaches to visual question answering, it achieves significant gains in data efficiency, compositional generalization, and transferability.

In Chapter 7, we introduce *metaconcepts*, which are concepts describing relations among other concepts. We present a meta-learning framework for learning new visual concepts quickly from just one or a few examples, guided by multiple naturally occurring data streams: simultaneously looking at images, reading sentences that describe the objects in the scene, and interpreting supplemental sentences that relate the novel concept to existing ones. The learned concepts support downstream applications such as answering questions about previously unseen images. This framework extends the neuro-symbolic paradigm to reason about higher-level relations among concepts, in addition to their grounding in observation.

In Chapter 8, we extend the NS-CL framework to multiple domains and modalities, such as 3D point clouds and motion capture trajectories. The key idea is to leverage large language models (LLMs) as pretrained, general-purpose models to translate natural language queries into executable programs in a shared first-order logic language. This chapter introduces a unified framework that learns to ground and reason with concepts across domains using a differentiable, domain-independent, first-order logic–based program executor. It demonstrates strong reasoning ability in a wide variety of tasks, including complex ones not seen during training, and can be readily applied to new domains.

Chapters 6–8 focus on learning visual concepts from paired visual observations and language. In contrast, in Chapter 9, we explore the reverse direction: how visual concept learning can facilitate language learning, such as acquiring the grammar rules of a new language. This framework unifies visual concept learning with word learning, enabling the joint acquisition of syntax and semantic forms of words and sentences. This addresses a key question in the broader concept learning framework: where do concepts come from? The answer, in this framework, is through the acquisition of word meanings. Moreover, it introduces a practical algorithm capable of generalizing from small amounts of data to novel word compositions in vision–language tasks.

Part III Part III builds on the concept learning and reasoning algorithms developed in Parts I and II, and presents studies on the representation and reasoning for actions and plans. Here, we investigate new representational and algorithmic tools for planning and control in environments that combine symbolic abstractions with continuous dynamics.

In particular, in Chapter 10, we introduce the *mechanism* representation for robot skills,

as well as its learning and reasoning algorithms. Each skill is represented as a sequence of basis manipulation operations, characterized by varying contact modes between robots and objects, such as grasping and releasing. This framework enables agents to learn skills from as few as 1 to 10 demonstrations, and to apply them compositionally to solve longer-horizon manipulation problems.

In Chapter 11, we present a framework for learning for long-horizon planning. It leverages two key ideas: neuro-symbolic visual concept learning introduced in Chapter 6, and neuro-symbolic skill representations introduced in Chapter 10, to build a planning-compatible representation of actions. In particular, in addition to representing the contact-mode-sequence structures of skills, we propose to also acquire the pre- and post-conditions of individual skills. The resulting framework illustrates how neuro-symbolic concepts can unify perception, reasoning, and action, enabling agents to operate flexibly and generalize reliably to novel sequences of primitive skills for novel states and goals.

In Chapter 12, we present a novel representational and reasoning framework that integrates all the key insights developed throughout the dissertation: neuro-symbolic logic reasoning, constraint satisfaction solving, concept learning, skill learning, and planning, with a particular focus on robotic planning problems. Specifically, we extend the planning-centric framework from Chapter 11 into a hybrid imperative–declarative decision-making framework. Rather than treating declarative planning and imperative policy execution as separate approaches, we unify them into a single representation. This design is inspired by the theoretical findings of Chapter 4 on reasoning about discrete actions, as well as the conceptual framework introduced in Chapter 5 for reasoning about continuous values. Taken together, this representation and reasoning framework opens up new directions for learning neuro-symbolic action representations and planning with them efficiently.

Taken together, these chapters sketch a unified neuro-symbolic approach, from theories to algorithms to systems, towards amchine intelligence, and they illustrate its effectiveness across several domains. Yet, much remains to be done. Future directions include tightening the integration of these methods into a single architecture capable of lifelong learning and reasoning, and extending the framework to support long-term memory, reasoning under partial observability and uncertainty, and more effective human–robot collaboration and alignment. In Chapter 13, we will discuss additional work needed to turn these components into a complete autonomous agent.

Part I

Foundations of Neuro-Symbolic Representation and Reasoning

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Neuro-Symbolic First-Order Logic

We begin our journey by looking into the problem of learning for logical reasoning. Logic, especially first-order logic with quantified rules, provides one of the most fundamental representational structures in AI, offering an abstract compositional way to describe objects, relations, and rules about the world. This chapter introduces the Neural Logic Machine (NLM), a neuro-symbolic architecture for learning rules from relational data. NLMs exploit the power of both neural networks—as function approximators, and logic programming—as a symbolic processor for objects with properties, relations, logic connectives, and quantifiers. After being trained on small-scale tasks (such as sorting short arrays), NLMs can recover lifted rules, and generalize to large-scale tasks (such as sorting longer arrays). This framework establishes a deep connection between first-order quantification structures in logic and neural network architectures; it also motivates the development of important techniques for differentiable program execution that will appear in later chapters, such as Chapter 6 and Chapter 11.

The material of this chapter has been previously published in Dong *et al.* [2019], which is a precursor to this thesis. This chapter also includes materials from Xiao *et al.* [2022]. Honghua Dong and Guangxuan Xiao, in particular, contributed significantly to the materials presented in this chapter.

2.1 Introduction

Deep learning has achieved great success in various applications such as speech recognition [Hinton *et al.*, 2012], image classification [Krizhevsky *et al.*, 2012; He *et al.*, 2015], machine translation [Sutskever *et al.*, 2014; Bahdanau *et al.*, 2015; Wu *et al.*, 2016; Vaswani *et al.*, 2017], and game playing [Mnih *et al.*, 2015; Silver *et al.*, 2017]. Starting from Fodor and Pylyshyn [1988], however, there has been a debate over the problem of systematicity (such as understanding recursive systems) in connectionist models [Fodor and McLaughlin, 1990;

Hadley, 1994; Jansen and Watter, 2012].

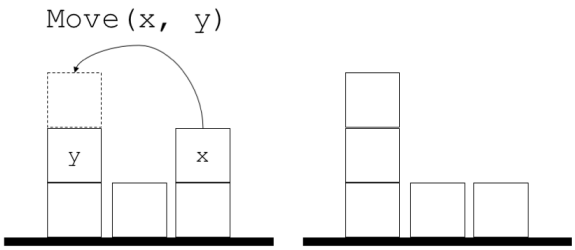
Logic systems can naturally process symbolic rules in language understanding and reasoning. Inductive logic programming (ILP) [Muggleton, 1991, 1996; Friedman *et al.*, 1999] has been developed for learning logic rules from examples. Roughly speaking, given a collection of positive and negative examples, ILP systems learn a set of rules (with uncertainty) that entails all of the positive examples but none of the negative examples. However, this is a fundamentally challenging task as the hypothesis space of possible rules is exponentially large [Dantsin *et al.*, 2001; Lin *et al.*, 2014a; Evans and Grefenstette, 2018].

To make the discussion concrete, let us consider the classic blocks-world problem [Nilsson, 1982; Gupta and Nau, 1992]. As shown in Fig. 2-1, we are given a set of blocks on the ground. We can move a block x and place it on the top of another block y or the ground, as long as x is *moveable* and y is *placeable*. We call this operation $\texttt{Move}(x, y)$. A block is said to be *moveable* or *placeable* if there are no other blocks on it. The ground is always *clear*, implying that we can place all blocks on the ground. Given an initial configuration of blocks-world, our goal is to transform it into a target configuration by taking a sequence of `Move` operations.

Although the blocks-world problem may appear simple at first glance, four major challenges exist in building a learning system to automatically accomplish this task:

1. The learning system should recover a set of lifted rules (i.e., rules that apply to objects generically instead of being tied to specific ones) and generalize to blocks-worlds which contain more blocks than those encountered during training. To get intuition about this, we refer the readers who are not familiar with the blocks-world domain to the task of learning to sort arrays [e.g., Vinyals *et al.*, 2015], where recurrent neural networks fail to generalize to arrays which are even just slightly longer than those for training.
2. The learning system should deal with high-order relational data and quantifiers, which goes beyond the scope of typical graph-structured neural networks [Kipf and Welling, 2017], which only consider binary relationships between objects. For example, to apply the transitivity rule of a relation r , i.e. $r(a, c) \leftarrow \exists b\ r(a, b) \wedge r(b, c)$, we need to jointly inspect three objects (a, b, c).
3. The learning system should scale up polynomially w.r.t. the complexity of the rules. Existing logic-driven approaches such as traditional ILP methods suffer an exponential computational complexity w.r.t. the number of logic rules to be learned [Dantsin *et al.*, 2001; Lin *et al.*, 2014a; Evans and Grefenstette, 2018].
4. The learning system should recover rules based on a minimal set of learning priors. In contrast, traditional ILP methods usually require hand-coded and task-specific rule templates to restrict the size of search spaces [Evans and Grefenstette, 2018].

In this chapter, we present Neural Logic Machines (NLMs) to address the aforementioned



$\texttt{On}(x, y)$	True if x is on y
$\texttt{IsGround}(x)$	True if x is the ground
$\texttt{Clear}(x)$	True if there is no block on x
$\texttt{Moveable}(x)$	$\neg\texttt{IsGround}(x) \wedge \texttt{Clear}(x)$
$\texttt{Placeable}(x)$	$\texttt{IsGround}(x) \vee \texttt{Clear}(x)$

Figure 2-1: (Top) A graphical illustration of the blocks world. Given an initial and a target worlds, the agent is required to move blocks to transform the initial configuration to the target one. (Bottom) A set of sentences used throughout the chapter to define the blocks world.

challenges. In a nutshell, NLMs are neuro-symbolic architectures that can realize first-order logic (FOL) rules in an arbitrary-sized domain. The key intuition behind NLMs is that logic operations such as logical ANDs and ORs can be realized with feed-forward neural networks, and the wiring among neural modules can realize the logic quantifiers, such as universal and existential quantifiers.

2.2 Neural Logic Machines (NLM)

The NLM is a neural realization of logic machines over finite domains and under the closed-world assumption*. Given a set of base predicates, grounded on a set of objects (the *premises*), NLMs sequentially apply first-order rules to draw *conclusions*, such as whether an object has a property. For example, in the blocks world, based on premises $\texttt{IsGround}(u)$ and $\texttt{Clear}(u)$ of object u, NLMs can infer whether u is moveable.

Internally, NLMs use tensors to represent logic predicates. This is done by grounding the predicate as `True` or `False` over a fixed set of objects. Based on the tensor representation, rules are implemented as neural operators that can be applied over the premise tensors and generate conclusion tensors. Such neural operators can handle relational data with various orders (i.e., operating on predicates with different arities).

*https://en.wikipedia.org/wiki/Closed-world_assumption

2.2.1 Logic Predicates as Tensors

We use tensors to represent the grounding of logic predicates. Suppose we have a universe of objects $\mathcal{U} = \{u_1, u_2, \ldots, u_m\}$. A predicate $p(x_1, x_2, \ldots, x_r)$, of *arity* r, can be grounded on the object set $\mathcal{U}$ (informally, we call it $\mathcal{U}$-*grounding*), resulting in a tensor $p^{\mathcal{U}}$ of shape $[m^r] \triangleq [m, m-1, m-2, \ldots, m-r+1]$, where the value of each entry $p^{\mathcal{U}}(u_{i_1}, u_{i_2}, \ldots, u_{i_r})$ of the tensor represents whether p is `True` under the grounding that $x_1 = u_{i_1}, x_2 = u_{i_2}, \cdots, x_r = u_{i_r}$. Here, we restrict that the grounded objects of all x_i's are mutually exclusive, i.e., $i_j \neq i_k$ for all pairs of indices j and k. This restriction does not limit the generality of the representation, as the "missing" entries can be represented by the $\mathcal{U}$-grounding of other predicates with a smaller arity. For example, for a binary predicate p, the grounded values of the $p^{\mathcal{U}}(x, x)$ can be represented by the $\mathcal{U}$-grounding of a unary predicate $p'(x) \triangleq p(x, x)$.

We extend this representation to a collection of predicates of the same arity. Let $C^{(r)}$ be the number of predicates of arity r. We stack the $\mathcal{U}$-*grounding* tensors of all predicates as a tensor of shape $\left[m^r, C^{(r)}\right] \triangleq \left[m, m-1, m-2, \ldots, m-r+1, C^{(r)}\right]$, where the last dimension corresponds to the predicates. Intuitively, a group of $C^{(1)}$ unary predicates grounded on m objects can be represented by a tensor of shape $\left[m, C^{(1)}\right]$, describing a group of "properties of objects", while a $\left[m, m-1, C^{(2)}\right]$-shaped tensor for $C^{(2)}$ binary predicates describes a group of "pairwise relations between objects". In practice, we set a maximum arity B for the predicates of interest, called the *breadth* of the NLM.

Each entry in $\mathcal{U}$-*grounding* tensors takes values in the range $[0, 1]$, which can be interpreted as a score indicating whether the predicate is `True`, grounded on a specific set of objects. All premises, conclusions, and intermediate results in NLMs are represented by such tensors. Such "softening" enables us to directly use gradient descent to optimize the weights. As a side note, we impose the restriction that all arguments in the predicates can only be variables or objects (i.e., constants) but not function symbols, which follows the setting of *Datalog* [Maier and Warren, 1988].

2.2.2 Logic Rules as Neural Operators

Our goal is to build a neural architecture to learn rules that are both lifted and able to handle relational data with multiple arities. We present different modules of our neural operators by making analogies to a set of essential *meta-rules* in symbolic logic systems. Specifically, we discuss our how our neural architecture can realize (1) *boolean logic rules*, as lifted rules containing boolean operations (`AND`, `OR`, `NOT`) over a set of predicates; and (2) *quantifications*, which bridge predicates with different arities by logic quantifiers ($\forall$ and $\exists$).

Next, we combine these neural units to compose NLMs. Figure 2-2 illustrates the overall multi-layer, multi-group architecture of an NLM. An NLM has layers of *depth D*

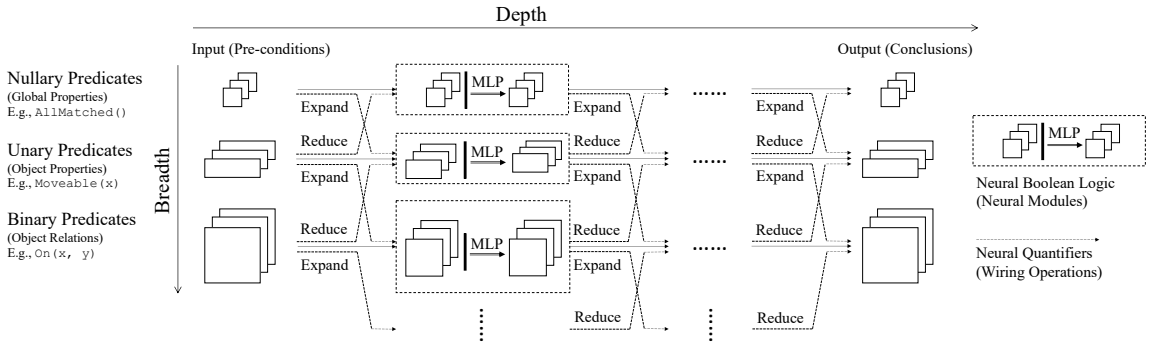


Figure 2-2: **An illustration of Neural Logic Machines (NLMs).** During forward propagation, NLM takes object properties and relations as input, performs sequential neural network computations, and outputs properties or relations of the objects. Implementation details can be found in Section 2.2.3.

(horizontally), and each layer has $B + 1$ computation units (vertically). These units operate on the tensor representations of predicates whose arities range from $[0, B]$, respectively. NLMs take input tensors of predicates (premises), perform layer-by-layer computations, and output tensors as conclusions.

As the number of layers increases, higher levels of abstraction can be formed. For example, the output of the first layer may represent $\texttt{Clear}(x)$, while a deeper layer may output a more complicated predicate like $\texttt{Moveable}(x)$. Thus, forward propagation in NLMs can be interpreted as a sequence of rule applications.

We start from the *neural boolean logic rules* and the *neural quantifiers*.

Boolean logic. We use the following symbolic meta-rule for boolean logic:

$$\hat{p}(x_1, x_2, \cdots, x_r) \leftarrow \texttt{expression}(x_1, x_2, \cdots, x_r), \tag{2.1}$$

where `expression` can be any *boolean* expressions consisting of predicates over *all* variables $(x_1, \ldots, x_r)$ and $\hat{p}(\cdot)$ is the derived predicate. For example, the rule $\texttt{Moveable}(x) \leftarrow \neg\texttt{IsGround}(x) \wedge \texttt{Clear}(x)$ can be instantiated from this meta-rule.

Denote $\mathcal{P} = \{p_1, \ldots, p_k\}$ as the set of $|\mathcal{P}|$ predicates appeared in `expression`. By definition, all p_i's have the same arity r and can be stacked as a tensor of shape $[m^r, |\mathcal{P}|]$. In Eq. 2.1, for a specific grounding of the derived predicate $\hat{p}(x_1 \cdots x_r)$, it is conditioned $r! \times |\mathcal{R}|$ grounding values with the same subset of objects, of *arbitrary permutation* as the arguments to all input predicates $\mathcal{P}$. For example, consider a specific ternary predicate $\hat{p}(x_1, x_2, x_3)$. For three different objects $a, b, c \in \mathcal{U}$, the grounding $\hat{p}(a, b, c)$ is conditioned on $p_j(a, b, c)$, $p_j(a, c, b)$, $p_j(b, a, c)$, $p_j(b, c, a)$, $p_j(c, a, b)$, $p_j(c, b, a)$ (all permutations of the

parameters) for all j (all input predicates).

Our neural implementation of boolean logic rules is a lifted neural module that uniformly applies to any grounding entries $(x_1 \cdots x_r)$ in the output tensor $\hat{p}^{\mathcal{U}}$. It has a Permute$(\cdot)$ operation transforming the tensor representation of $\mathcal{P}$, followed by a multi-layer perceptron (MLP). Given the tensor representation of $\mathcal{P}$, for each $p_i^{\mathcal{U}}(x_1, x_2, \ldots, x_r)$, the Permute$(\cdot)$ operation creates $r!$ new tensors as $p_{i,1}^{\mathcal{U}}, \ldots, p_{i,r!}^{\mathcal{U}}$ by permuting all axes that index objects, with all possible permutations. We stack all to form a $[m^{\underline{r}}, r! \times |\mathcal{P}|]$-shaped tensor. An MLP uniformly applies to all $m^{\underline{r}}$ object indices:

$$\hat{p}(u_{i_1}, \cdots, u_{i_r}) = \sigma\left(\text{MLP}\left(p_{1,1}(u_{i_1}, \ldots, u_{i_r}), \cdots, p_{k,r!}(u_{i_1}, \ldots, u_{i_r})\right); \theta\right), \tag{2.2}$$

where σ is the sigmoid nonlinearity, θ is the trainable network parameters. For all sets of mutually exclusive indexes $i_1, \ldots, i_r \in \{1, 2, \ldots, m\}$, the same MLP is applied. Thus, the size of θ is independent of the number of objects m. This property is analogous to the implicit unification property of Horn clauses: the rule $\hat{p}(x) \leftarrow p_1(x) \wedge p_2(x)$ implicitly means, $\forall x\ \hat{p}(x) \leftarrow p_1(x) \wedge p_2(x)$.

Quantification. We introduce two types of meta-rules for quantification, namely *expansion* and *reduction*. Let p be a predicate, and we have

(Expansion) $$\forall x_{r+1}\ q(x_1, x_2, \cdots, x_r, x_{r+1}) \leftarrow p(x_1, x_2, \cdots, x_r), \tag{2.3}$$

where $x_{r+1} \notin \{x_i\}_{i=1}^{r}$. The expansion operation constructs a new predicate q from p, by introducing a new variable x_{r+1}. For example, consider the following rule

$$\texttt{ValidMove}(x, y) \leftarrow \texttt{Moveable}(x) \wedge \texttt{Placeable}(y).$$

This rule does not fit the meta-rule in Eq. 2.1 as some predicates on the RHS only take a *subset* of variables as inputs. However, it can be described by using the expansion and the boolean logic meta-rules jointly.

1. $\forall z\ \texttt{MoveableX}(x, z) \leftarrow \texttt{Moveable}(x)$;	(from Eq. 2.3)
2. $\forall z\ \texttt{PlaceableY}(y, z) \leftarrow \texttt{Placeable}(y)$;	(from Eq. 2.3)
3. $\texttt{ValidMove}(x, y) \leftarrow \texttt{MoveableX}(x, y) \wedge \texttt{PlaceableY}(y, x)$.	(from Eq. 2.1)

The *expansion meta-rule* (Eq. 2.3) for a set of C r-ary predicates, represented by a $[m^{\underline{r}}, C]$-shaped tensor, introduces a new and distinct variable x_{r+1}. Our neural implementation Expand$(\cdot)$ repeats each predicate (their tensor representation) for $(m - r)$ times, and stacks in a new dimension. Thus the output shape is $[m^{\underline{r+1}}, C]$.

The other meta-rule is for reduction:

$$\textbf{(Reduction)} \qquad q(x_1, x_2, \cdots, x_r) \leftarrow \forall x_{r+1}\ p(x_1, x_2, \cdots, x_r, x_{r+1}), \tag{2.4}$$

where the $\forall$ quantifier can also be replaced by $\exists$. The reduction operation reduces a variable in a predicate via the quantifier. As an example, the rule to deduce the moveability of objects,

$$\texttt{Moveable}(x) \leftarrow \neg\texttt{IsGround}(x) \wedge \neg(\exists y\ \texttt{On}(y, x)),$$

can be expressed using meta-rules as follows:

1. $\texttt{Clear}(x) \leftarrow \forall y\ \neg\texttt{On}(y, x)$; (from Eq. 2.4)
2. $\texttt{Moveable}(x) \leftarrow \neg\texttt{IsGround}(x) \wedge \texttt{Clear}(x)$. (from Eq. 2.1)

The *reduction meta-rule* (Eq. 2.4) for a set of C $(r+1)$-ary predicates, represented by a $[m^{\underline{r+1}}, C]$-shaped tensor, eliminates the variable x_{r+1} via quantifiers. For $\exists$ (or $\forall$), our neural implementation Reduce$(\cdot)$ takes the maximum (or minimum) element along the dimension of x_{r+1}, and stacks the two resulting tensors. Therefore, the output shape becomes $[m^{\underline{r}}, 2C]$.

2.2.3 Neural Logic Machines

NLMs realize symbolic logic rules in a multi-layer multi-group architecture, illustrated in Fig. 2-2. An NLM has D layers, and each layer has $B+1$ computation units as groups. Between layers, we use *intra-group computation* (Eq. 2.1). The predicates at each layer are grouped by their arities, and inside each group, we use *inter-group computation* (Eq. 2.3 and 2.4).

We define $O_i = \left\{O_i^{(0)}, O_i^{(1)}, \cdots, O_i^{(B)}\right\}$ as the outputs of layer i, where $O_i^{(r)}$ is the output corresponding to the r-ary unit at layer i. For convenience, we denote $O_0 = \left\{O_0^{(0)}, O_0^{(1)}, \cdots, O_0^{(B)}\right\}$ as the $\mathcal{U}$-grounding tensors for NLM's base predicates (the *premises*), and O_D at the last layer as the *conclusions*. The overall computation is performed layer-by-layer, from layer 1 to layer D. All computation units at layer i work simultaneously, taking O_{i-1} as inputs and generating O_i.

Let us consider a specific group r at layer i, and we show how to calculate $O_i^{(r)}$.

Inter-group computation. As shown in Figures 2-2 and 2-3, we connect tensors from the previous layer $i-1$ in vertically neighboring groups (i.e. $r-1$, r and $r+1$), and aligns their shapes by *expansion* (Eq. 2.3) or *reduction* (Eq. 2.4) to form an intermediate tensor $I_i^{(r)}$:

$$I_i^{(r)} = \text{Concat}\left(\text{Expand}\left(O_{i-1}^{(r-1)}\right), O_{i-1}^{(r)}, \text{Reduce}\left(O_{i-1}^{(r+1)}\right)\right). \tag{2.5}$$

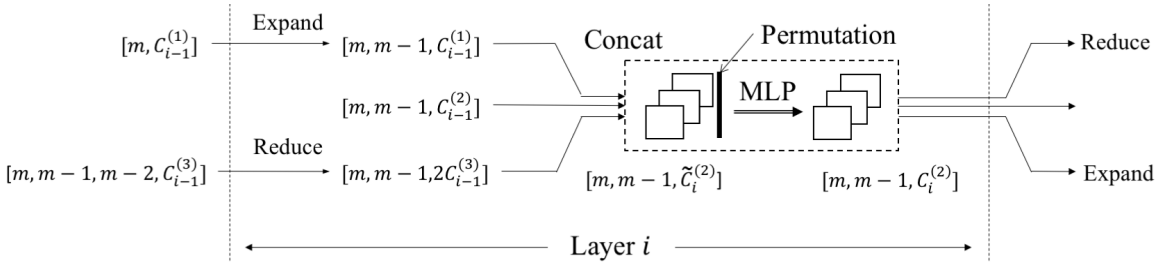


Figure 2-3: **An illustration of the computational block inside NLM for binary predicates at layer i.** $C_i^{(j)}$ denotes the number of output predicates of group j at layer i. $[\cdot]$ denotes the shape of the tensor.

Nonexistent terms are ignored (e.g. when $r+1 > B$ or $r-1 < 0$). Note that from the previous layer, $O_{i-1}^{(r-1)}, O_{i-1}^{(r)}, O_{i-1}^{(r+1)}$ have shapes $[m^{\underline{r-1}}, C_{i-1}^{(r-1)}]$, $[m^{\underline{r}}, C_{i-1}^{(r)}]$, $[m^{\underline{r+1}}, C_{i-1}^{(r+1)}]$, respectively. After the concatenation, the resulting tensor $I_i^{(r)}$ is of shape $[m^{\underline{r}}, \widetilde{C}_i^{(r)}]$, where the number of new predicates is $\widetilde{C}_i^{(r)} \triangleq C_{i-1}^{(r-1)} + C_{i-1}^{(r)} + 2C_{i-1}^{(r+1)}$, and the 2 comes from the two quantifiers ($\forall$ and $\exists$).

The inter-group computation essentially aligns predicates of neighboring arities. Relational representations of different orders get combined together through the neural quantification.

Intra-group computation. The intra-group computation is implemented as the neural boolean logic in Eq. 2.1. It take the intermediate tensor $I_i^{(r)}$ as input, permutes and generates the output tensor $O_i^{(r)}$:

$$O_i^{(r)} = \sigma\left(\text{MLP}\left(\text{Permute}\left(I_i^{(r)}\right); \theta_i^{(r)}\right)\right), \tag{2.6}$$

where σ is the sigmoid nonlinearity and $\theta_i^{(r)}$ denotes trainable parameters. We apply Permute function to $\widetilde{C}_i^{(r)}$ tensors in $I_i^{(r)}$ individually, and get $r!\widetilde{C}_i^{(r)}$ tensors. We set the number of output neurons to be $C_i^{(r)}$, thus the shape of output tensor $O_i^{(r)}$ is $[m^{\underline{r}}, C_i^{(r)}]$.

Example. For concreteness, in Fig. 2-3, consider group 2 (*binary* predicates) at layer i. The module begins with the inter-group computation. It first collects the output of vertically consecutive groups (unary, binary and ternary) from the previous layer $i-1$, where their shapes are shown in the figure. Then it uses expansion/reduction to compose the intermediate tensor $I_i^{(2)}$ containing $\widetilde{C}_i^{(2)} \triangleq C_{i-1}^{(1)} + C_{i-1}^{(2)} + 2C_{i-1}^{(3)}$ predicates. For each object pair (x, y), the output $\mathcal{U}$-grounding tensor of predicates is computed by intra-group computation $O_i^{(2)}(x,y) = \text{MLP}(\text{Concat}(I_i^{(2)}(x,y), I_i^{(2)}(y,x)); \theta_i^{(2)})$, and the output shape is $[m, m-1, C_i^{(2)}]$. The $\text{Concat}(\cdot,\cdot)$ corresponds to the `Permute` operation, while the MLP is shared among all pairs of objects (x, y).

Remark. It can be verified that NLMs can realize forward chaining of a set of Horn clauses with only finitely many entities and finite-depth chaining. Thus, there should not exist cyclic references of predicates among rules. The extension to support cyclic references is left as future work. Thus, given the training dataset containing pairs of (*premises*, *conclusions*), NLMs can induce lifted rules that entail the *conclusions* and generalize w.r.t. the number of objects during testing.

2.2.4 Expressiveness and Computational Complexity

We will present a formal characterization of the expressive power of NLMs in Chapter 4. Here, we summarize a few key realizability results.

1. The *depth* D of NLM (i.e., number of layers) restricts the maximum number of deduction steps.
2. The *breadth* B of NLM (i.e., the maximum number of variables in all predicates considered) limits the arity of relations among objects. Practically, most (intermediate) predicates are binary or ternary and we set B depending on the task (typically 2 or 3).
3. The number of output predicates used at each layer ($C_i^{(r)}$ in Fig. 2-3). Let $C = \max_{i,r} C_i^{(r)}$, and this number is often small in our experiments (e.g., 8 or 16).
4. In Eq. 2.2, the expressive power of MLP (number of hidden layers and number of hidden neurons) restricts the complexity of the boolean logic to be represented. In our experiments, we usually prefer shallow networks (e.g., 0 or 1 hidden layer) with a small number of neurons (e.g., 8 or 16). This can be viewed as a low-dimension regularization on the logic complexity and encourages the learned rule to be simple.

The computational complexity of NLM's forward or backward propagation is $O(m^B DC^2)$ where m is the number of objects. The network has $O(DC^2)$ parameters, which is independent of the number of objects. Assuming B is a small constant, the computational complexity of NLM is quadratic in the number of allowed predicates.

2.3 Experiments

In this section, we show that NLM can solve a broad set of tasks, ranging from relational reasoning to decision making. Furthermore, we show that NLM trained using small-sized instances can generalize to large-sized instances. In the experiments, Softmax-Cross-Entropy loss is used for supervised learning tasks, and REINFORCE [Williams, 1992] is used for reinforcement learning tasks.

Due to space limitation, interested readers are referred to Appendix A.1 for details of training (including curriculum learning) in the decision making tasks, and Appendix A.2

for more implementation details (such as residual connections [He *et al.*, 2015]), hyper-parameters, and model selection criterion.

2.3.1 Baselines

We consider two baselines as representatives of the connectionist and symbolicist: Memory Networks (MemNN) [Sukhbaatar *et al.*, 2015] and Differentiable Inductive Logic Programming (∂ILP) [Evans and Grefenstette, 2018], a state-of-the-art ILP framework. We also make comparisons with other models such as Differentiable Neural Computer (DNC) [Graves *et al.*, 2016] and graph neural networks [Li *et al.*, 2016] whenever eligible.

For MemNN, in order to handle an arbitrary number of inputs (properties, relations), we adopt the method from Graves *et al.* [2016]. Specifically, each object is assigned with a unique identifier (a binary integer ranging from 0 to 255), as its "name". The memory of MemNN is now a set of "pre-conditions". For unary predicates, the memory slot contains a tuple $(\texttt{id}(x), 0, \texttt{properties}(x))$ for each x, and for binary predicates $p(x, y)$, the memory slot contains a tuple $(\texttt{id}(x), \texttt{id}(y), \texttt{relations}(x, y))$, for each pair of (x, y). Both $\texttt{properties}(x)$ and $\texttt{relations}(x, y)$ are length-k vectors v, where k is the number of input predicates. We number each input predicate with an integer $i = 1, 2, \cdots, k$. If object x has a property $p_i(x)$, then $v[i] = 1$; otherwise, $v[i] = 0$. If a pair of objects (x, y) have relation $p_i(x, y)$, then $v[i] = 1$; otherwise, $v[i] = 0$. We extract the key and value for MemNN's to lookup on the given pre-conditions with 2-layer multi-layer perceptrons (MLP). MemNN relies on iterative queries to the memory to perform relational reasoning. Note that MemNN takes a sequential representation of the multi-relational data.

For ∂ILP, the grounding of all base predicates is used as the input to the system.

2.3.2 Family Tree Reasoning

The family tree is a benchmark for inductive logic programming, where the machine is given a family tree containing m members. The family tree is represented by the following relations (predicates): `IsSon`, `IsDaughter`, `IsFather` and `IsMother`. The goal of the task is to reason out other properties of family members or relations between them. Our results are summarized in Table 2.1.

For MemNN, we treat the problem of relation prediction as a question answering task. For example, to determine whether member x has a father in the family tree, we input $\texttt{id}(x)$ to MemNN as the question. MemNN then performs multiple queries to the memory and updates its hidden state. The finishing hidden state is used to classify whether $\texttt{HasFather}(x)$. For relations (binary predicates), the corresponding MemNN takes the concatenated embedding of $\texttt{id}(x)$ and $\texttt{id}(y)$ as the question.

Table 2.1: **Results on family tree and graph reasoning.** Comparison among MemNN, ∂ILP and the proposed NLM in family tree and graph reasoning, where m is the size of the testing family trees or graphs. Both ∂ILP and NLM outperform the neural baseline and achieve perfect accuracy (100%) on test set. Note N/A mark means that ∂ILP cannot scale up in 2-`OutDegree`.

Family Tree	MemNN		∂ILP		NLM (Ours)	
	$m = 20$	$m = 100$	$m = 20$	$m = 100$	$m = 20$	$m = 100$
`HasFather`	99.9% / 99.9%	59.8% / 65.2%	100%	100%	100%	100%
`HasSister`	86.3% / 85.5%	59.8% / 66.4%	100%	100%	100%	100%
`IsGrandparent`	96.5% / 84.7%	97.7% / 63.7%	100%	100%	100%	100%
`IsUncle`	96.3% / 85.8%	96.0% / 64.0%	100%	100%	100%	100%
`IsMGUncle`	99.7% / 98.4%	98.4% / 81.7%	100%	100%	100%	100%
Graph	MemNN		∂ILP		NLM (Ours)	
	$m = 10$	$m = 50$	$m = 10$	$m = 50$	$m = 10$	$m = 50$
`AdjacentToRed`	95.2% / 94.6%	93.1% / 91.9%	100%	100%	100%	100%
4-`Connectivity`	92.3% / 90.5%	81.3% / 88.0%	100%	100%	100%	100%
6-`Connectivity`	67.6% / 58.8%	43.9% / 67.9%	100%	100%	100%	100%
1-`OutDegree`	99.8% / 99.7%	78.6% / 81.2%	100%	100%	100%	100%
2-`OutDegree`	81.4% / 61.8%	96.7% / 87.7%	N/A	N/A	100%	100%

For ∂ILP, we take the grounded probability of the "target" predicate as the output; for an NLM with D layers, we take the corresponding group of output predicates at the last layer (for property prediction, we use tensor $O_D^{(1)}$ to represent unary predicates, while for relation prediction we use tensor $O_D^{(2)}$ to represent binary predicates) and classify the property or relation with a linear layer.

All models are trained on instances of size 20 and tested on instances of size 20 and 100 (size is defined as the number of family members). The models are trained with fully supervised learning (labels are available for all objects or pairs of objects). During the testing phase, the accuracy is evaluated (and averaged) on all objects (for properties such as `HasFather`) or pairs of objects (for relations such as `IsUncle`). `MGUncle` is defined as one's maternal great uncle, which is also used by Differentiable Neural Computer (DNC) [Graves *et al.*, 2016]. We report the performance of MemNN in the format of Micro / Macro accuracy. We also try our best to replicate the setting used by Graves *et al.* [2016], and as a comparison, in the task of "finding" the `MGUncle` instead of "classifying", DNC reaches the accuracy of 81.8%.

2.3.3 General Graph Reasoning

We further extend the Family tree to general graphs and report the reasoning performance in Table 2.1.

We treat each node in the graph as an object (symbol). The (undirected) graph is fed into the model in the form of a "`HasEdge`" relation between nodes (which is an adjacent matrix). Besides, an extra property `color` represented by one-hot vectors is defined for every node. A node has the property of `AdjacentToRed` if it is adjacent to a red node by an outgoing edge. k-`Connectivity` is a relation between two nodes in the graph, which is true if two nodes are connected by a path with length at most k. A node has property k-`OutDegree` if its out-degree is exactly k. The `N/A` result of ∂ILP in the 2-`OutDegree` task comes from its memory restriction [Evans and Grefenstette, 2018], where 3-ary intentional predicate is required. As an example, a human-written logic rule for 2-`OutDegree` can be $\texttt{-OutDegree}(a) \leftarrow \exists_b \exists_c \forall_d \texttt{HasEdge}(a, b) \wedge \texttt{HasEdge}(a, c) \wedge \neg \texttt{HasEdge}(a, d)$ where a, b, c and d are distinct nodes in the graph.

All models are trained on instances of size 10 and tested on instances of size 10 and 50 (size is defined as the number of nodes in the graph).

2.3.4 Blocks World

We also test NLM's capability of decision making in the classic blocks world domain [Nilsson, 1982; Gupta and Nau, 1992] by slightly extending the model to fit the formulation of Markov Decision Process (MDP) in reinforcement learning.

Shown in Fig. 2-1, an instance of the blocks world environment contains two worlds: the initial world and the target world, each containing the ground and m blocks. The task is to take actions in the operating world and make its configuration the same as the target world. The agent receives positive rewards only when it accomplishes the task and the sparse reward setting brings significant hardness. Each object (blocks or ground) can be represented by four properties: `world_id`, `object_id`, `coordinate_x`, `coordinate_y`. The ground has a fixed coordinate $(0, 0)$. The input is the result of the numeral comparison among all pairs of objects (may come from different worlds). For example, in x-coordinate, the comparison produces three relations for each object pair $(i, j), i \neq j$: $\texttt{Left}(i, j)$ (whether i is to the left of j, or $\mathbf{1}[x_i < x_j]$), $\texttt{SameX}(i, j)$ and $\texttt{Right}(i, j)$.

The only operation is $\texttt{Move}(i, j)$, which moves object i onto the object j in the operating world if i is movable and j is placeable. If the operation is invalid, it will have no effect; otherwise, the action takes effect and the state represented as coordinates will change accordingly. In our setting, an object i is movable iff it is not the ground and there are no blocks on it, i.e. $\forall_j \neg(\texttt{Up}(i, j) \wedge \texttt{SameX}(i, j))$. Object i is placeable iff it is the ground or there

Task	MemNN		NLM (Ours)	
	$m = 10$	$m = 50$	$m = 10$	$m = 50$
`BlocksWorld`	0% / N/A	0% / N/A	100% / 12	100% / 84
`Sorting`	100% / 22	90% / 986.6	100% / 8	100% / 45
`Path`	45% / 13.3	12% / 42.7	100% / 4	100% / 4

Table 2.2: **Results on blocks world and algorithmic reasoning tasks.** Comparison between MemNN and the proposed NLM in the blocks world, sorting integers, and finding shortest paths, where m is the number of blocks in the blocks world environment or the size of the arrays/graphs in sorting/path environment. Both models are trained on instance size $m \leq 12$ and tested on $m = 10$ or 50. The performance is evaluated by two metrics and separated by "/": the probability of completing the task during the test, and the average `Moves` used by the agents when they complete the task. There is no result for ∂ILP since it fails to scale up. MemNN fails to complete the blocks world within the maximum $m \times 4$ `Moves`.

are no blocks on it.

To avoid the ambiguity of the x-coordinates while putting blocks onto the ground, we set the x coordinate of block i to be i when it is placed onto the ground. The action space is $(m+1)\times m$ where m is the number of blocks in the world and $+1$ comes from the "ground". For both MemNN and NLM, we apply a shared MLP on the output relational predicates of each pair of objects $O_D^{(2)}(x, y)$ and compute an action score $s(x, y)$. The probability for `Move`(x, y) is $\propto \exp s(x, y)$ (by taking a `Softmax`). The results are summarized in Table 2.2. For more discussion on the confidence bounds of the experiments, please refer to Appendix A.2.6.

2.3.5 General Algorithms

We further show NLM's ability to excel at algorithmic tasks, such as sorting number arrays (`Sorting`) and path finding (`Path`). We view an algorithm as a sequence of primitive actions and cast as a reinforcement learning problem.

Sorting. We first consider the problem of sorting integers. Given a length-m array a of integers, the algorithm needs to iterative swap elements to sort the array in ascending order. We treat each slot in the array as an object, and input their index relations (whether $i < j$) and numeral relations (whether $a[i] < a[j]$) to NLM or MemNN. The action space is $m \times (m-1)$ indicating the pair of integers to be swapped. Table 2.2 summarizes the learning performance.

As the comparisons between all pairs of elements in the array are given to the agent, sorting the array within the maximum number of swaps is an easy task. A trivial solution is

to randomly swap an inversion[†] in the array at each step.

Beyond being able to generalize to arrays of arbitrary length, with different hyper-parameters and random seeds, the learned algorithms can be interpreted as Selection-Sort, Bubble-Sort, etc. We include videos demonstrating some learned algorithms in our website.[‡]

Path finding. We also test the performance of finding a path (single-source single-target path) in a given graph as a sequential decision-making problem in reinforcement learning environment. Given an undirected graph represented by its adjacency matrix as relations, the algorithm needs to find a path from a start node s (with property `IsStart`(s) = `True`) to the target node t (with property `IsTarget`(t) = `True`). To restrict the number of deduction steps, we set the maximum distance between s and t to be 5 during the training and set the distance between s and t to be 4 during the testing, which replicates the setting of Graves *et al.* [2016]. Table 2.2 summarizes the result.

`Path` task here can be seen as an extension of bAbI task 19 (path finding) [Weston *et al.*, 2015] with symbolic representation. As a comparison with graph neural networks, Li *et al.* [2016] achieved 99% accuracy on the bAbI task 19. Contrastively, we formulate the shortest path task as a more challenging reinforcement learning (decision-making) task rather than a supervised learning (prediction) task as in Graves *et al.* [2016]. Specifically, the agent iteratively chooses the next node *next* along the path. At the next step, the starting node will become *next* (at each step, the agent will move to *next*). As a comparison, in Graves *et al.* [2016], Differentiable Neural Computer (DNC) finds the shortest path with probability 55.3% in a similar setting.

2.4 Sparsity and Locality in Neural Logic Machines

The neural logic machine framework has enabled us to reason about the relationships between entities from input facts (e.g., whether Ari is a *grandparent* of Charlie). This generally requires explicit consideration of other entities that are not mentioned in the query (e.g., the *parents* of Charlie). However, due to its inherent "dense" nature, it is not readily scalable to problems of large sizes. In this section, we will present an approach for learning to solve problems of this kind in large, real-world domains, using *sparse and local hypergraph neural networks* (SpaLoc). SpaLoc achieves state-of-the-art performance on several real-world, large-scale knowledge graph reasoning benchmarks, and is the first framework for applying hypergraph neural networks on real-world knowledge graphs with more than 10k nodes.

[†] https://en.wikipedia.org/wiki/Inversion_(discrete_mathematics)
[‡] https://sites.google.com/view/neural-logic-machines

Concretely, consider the problem of learning to predict the *grandparent* relationship. Given a dataset of labeled family relationship graphs, we aim to build machine-learning algorithms that learn to predict a specific relationship (e.g., *grandparent*) based on other input relationships, such as *father* and *mother*. A crucial feature of such reasoning tasks is that: in order to predict the relationship between two entities (e.g., whether Ari is a *grandparent* of Charlie), we need to jointly consider other entities (e.g., the *father* and *mother* of Charlie).

NLMs can be viewed as *hypergraph neural networks* representing and reasoning about higher-order relations. In a hypergraph, a *hyper-edge* may connect more than two nodes. In particular, NLMs maintain hyperedge representations for all tuples consisting of up to B entities, where B is a hyperparameter. Thus, they can infer more complex finitely-quantified logical relations than standard graph neural networks that only consider binary relationships between entities [Morris *et al.*, 2019; Barceló *et al.*, 2020]. However, there are two disadvantages of such a dense hypergraph representation. First, the training and inference require considering all entities in a domain simultaneously, such as all of the N people in a family relationship database. Second, they scale exponentially with respect to the number of entities considered in a single type of relationship: inferring the *grandparent* relationship between all pairs of entities requires $O(N^3)$ time and space complexity. In practice, for large graphs, these limitations make the training and inference intractable and hinder the application of methods such as NLMs in large-scale real-world domains.

To address these two challenges, we draw two inspirations from traditional logic-based reasoning: logical rules (e.g., my parent's parent is my grandparent) usually apply *locally* (e.g., only three people are involved in a grandparent rule), and *sparsely* (e.g., the grandparent relationship is sparse across all pairs of people in the world). Thus, during training and inference, we don't need to keep track of the representation of *all* hyperedges but only the hyperedges that are related to our prediction tasks.

Inspired by these observations, we develop the Sparse and Local Hypergraph Neural Network (SpaLoc) for learning sparse relational representations from data in large domains. First, we present a *sparse* tensor-based representation for encoding hyperedge relationships among entities and extend hypergraph neural networks to this representation. Instead of storing a dense representation for all hyperedges, it only keeps track of edges related to the prediction task, which exploits the inherent sparsity of hypergraph reasoning. Second, since we do not know the underlying sparsity structures *a priori*, we propose a training paradigm to recover the underlying *sparse* relational structure among objects by regularizing the graph sparsity. During training, the graph sparsity measurement is used as a soft constraint, while during inference, SpaLoc uses it to explicitly prune out irrelevant edges to accelerate the inference. Third, during both training and inference, SpaLoc focuses on a *local* induced

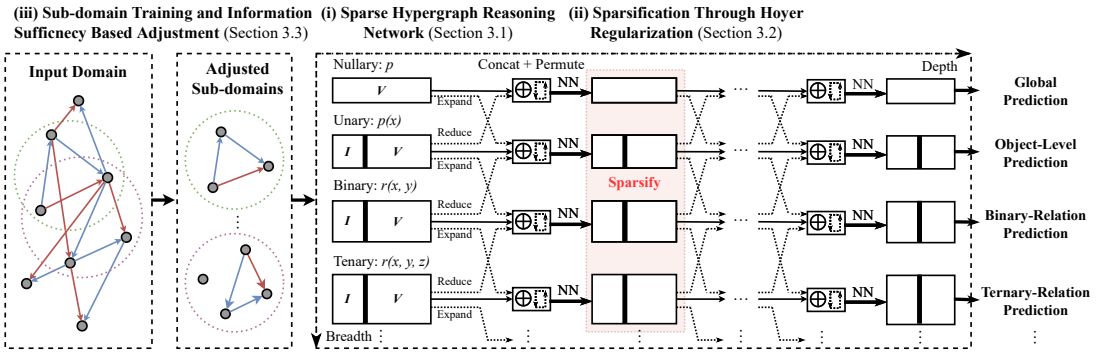


Figure 2-4: **The overall training pipeline of SpaLoc**. It includes sub-graph sampling with label adjustment (Sec. 2.4.3), sparse hypergraph neural networks (Sec. 2.4.1), and sparsity regularizations (Sec. 2.4.2). I and V denote the index tensor and value tensor, respectively.

subgraph of the input graph, instead of considering all entities and their relations. This is achieved by a novel sub-graph sampling technique motivated by *information sufficiency* (IS). IS quantifies the amount of information in a sub-graph for predictions about a specific hyperedge. Since the information in a sub-sampled graph may be insufficient for predicting the relationship between a pair of entities, we also propose to use the information sufficiency measure to adjust training labels.

As illustrated in Fig. 2-4, we make hypergraph networks practical for large domains by using sparse tensors (Section 2.4.1). To encourage models to discover sparse interconnections, we add sparsity regularization to intermediate tensors (Section 2.4.2). We exploit the locality of the task by sampling subgraphs and compensate for information loss due to sampling through a novel label adjustment process (Section 2.4.3).

The fundamental structures used for both training and inference are hypergraphs $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of vertices and $\mathcal{E}$ is a set of hyperedges. Each hyperedge $e = (x_1, x_2, \cdots, x_r)$ is an ordered tuple of r elements (r is called the arity of the edge), where $x_i \in \mathcal{V}$. We use $f : \mathcal{E} \to \mathcal{S}$ to denote a *hyperedge representation function*, which maps hyperedge e to a feature in $\mathcal{S}$. Domains $\mathcal{S}$ can be of various forms, including discrete labels, numbers, and vectors. For simplicity, we describe features associated with arity-1 edges as "node features" and features associated with the whole graph as "nullary" or "global" features.

In this section, we consider graph-reasoning tasks that can be formulated as follows: given $\mathcal{H}$ and the input hyperedge representation functions f associated with all hyperedges in $\mathcal{E}$, such as node types and pairwise relationships (e.g., *parent*), our goal is to infer a target representation function f' for one or more hyperedges, i.e. $f'(e)$ for some $e \in \mathcal{E}$, such as predicting a new relationship (e.g., *grandparent(Kim,Skye)*). We consider two problem settings in this section. The first one is to predict a target relation over all edges in the graph. The second one is to predict the relation on one single edge.

2.4.1 Sparse Hypergraph Neural Networks

SpaLoc is a general formulation that can be applied to a range of hypergraph reasoning frameworks. We will primarily develop our method based on NLMs. We choose an NLM as the backbone network in SpaLoc because its tensor representation naturally generalizes to sparse cases. In Section 2.4.4, we also integrate SpaLoc with other hypergraph neural networks like k-GNNs [Morris *et al.*, 2019].

In SpaLoc, hypergraph features such as node features and edge features are represented as sparse tensors. For example, as shown in Fig. 2-4, at the input level, the parental relationship can be represented as a list of indices and values. In this case, each index (x, y) is an ordered pair of integers, and the corresponding value is 1 if node x is a parent of node y. To leverage the sparsity in relations, we treat values for indices not in the list as 0. This convention also extends to vector representations of nodes and hyperedges. In general, vector representations $f(x_1, x_2, \cdots, x_r)$ associated with all hyperedges of arity r are represented as coordinate-list (COO) format sparse tensors [Fey and Lenssen, 2019]. That is, each tensor is represented as two tensors $\mathcal{F} = (\mathbf{I}, \mathbf{V})$, each with M entries. The first tensor $\mathbf{I}$ is an *index* tensor, of shape $M \times r$, in which each row denotes a tuple $(x_1, x_2, \cdots, x_r)$. The second tensor $\mathbf{V}$ is a *value* tensor, of shape $M \times D$, where D is the length of $f(x_1, x_2, \cdots, x_r)$. Each row $\mathbf{V}[i]$ denotes the vector representation associated with the tuple $\mathbf{I}[i]$. For all tuples that are not recorded in $\mathbf{I}$, their representations are treated as all-zero vectors.

Based on the sparse feature representations, a sparse hypergraph neural network is composed of multiple *relational reasoning layers* (RRLs) that operate on hyperedge representations. Fig. 2-4 shows the detailed computation graph of a SpaLoc model with ternary relations. The input to first RRL is the input information (e.g., demographic information and parental relationships in a person database). Each RRL computes a set of new hyperedge features as inputs to the next layer. The last layer output will be the final prediction of the task (e.g., the "son" relationship). During training time, we will supervise the network with ground-truth labels for final predictions.

Next, we describe the computation of individual RRLs. The descriptions will be brief and focus on differences from the original NLM layers. The input to and output of each RRL are both $R + 1$ sparse tensors of different arities, where R is the maximum arity of the network. Let $\mathcal{F}^{(i-1,r)}$ denote the input of arity r of layer i, the output of this layer $\mathcal{F}^{(i,r)}$ is computed as the following:

$$\mathcal{F}^{(i,r)} = \mathrm{NN}^{(i,r)}\left(\textsc{permute}\left(\textsc{concat}\left(\mathcal{F}^{(i-1,r)}, \textsc{expand}\left(\mathcal{F}^{(i-1,r-1)}\right), \textsc{reduce}\left(\mathcal{F}^{(i-1,r+1)}\right)\right)\right)\right)$$

In a nutshell, the EXPAND operation propagates representations from lower-arity tensors to a higher-arity form (e.g., from each node to the edges connected to it). The REDUCE operation aggregates higher-arity representations into a lower-arity form (e.g., aggregating

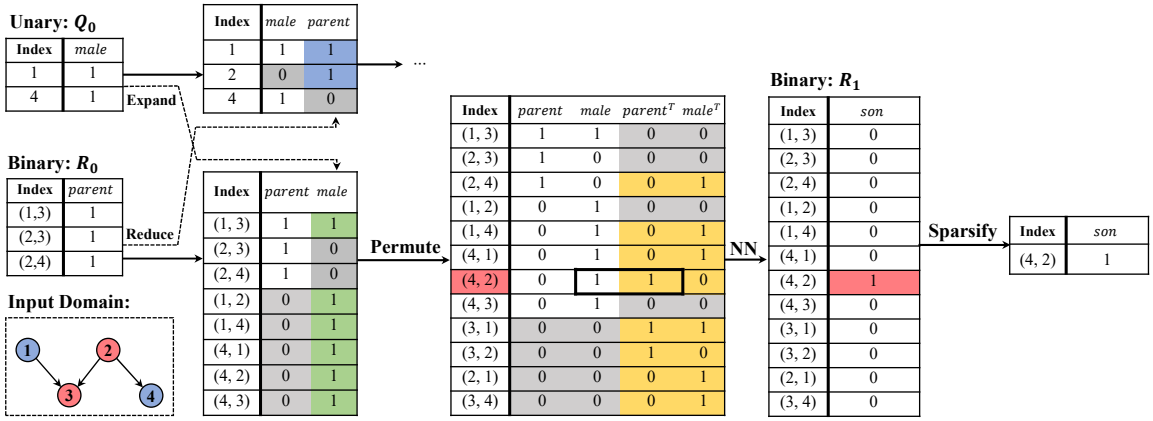


Figure 2-5: **A running example of a single-layer SpaLoc.** It infers the binary relationship of *son(x, y)* := *male(x)* ∧ *parent(y, x)* from the attribute *male* and the binary relationship *parent*. The model first expands the unary tensor (containing the *male* information) into a binary relation, indicating whether the first entity in the pair is a male. Then, the permutation operation fuses the information for (x, y) and (y, x). For each pair (x, y), we now have four predicates: whether x is a parent of y, whether y is a parent of x, whether x is a male, and whether y is a male. Finally, a neural network predicts the target relationship *son* for each pair (x, y). Blue entries denote values that are reduced from high-arity tensors. Green entries are expanded from low-arity tensors. Yellow entries are created by the "permutation" operation. Gray entries are zero paddings.

the information from all edges connected to a node into that node). The PERMUTE operation fuses the representations of hyperedges that share the same set of entities but in different orders, such as (A, B) and (B, A). Finally, NN is a linear layer with nonlinear activation that computes the representation for the next layer. Fig. 2-5 gives a concrete running example of a single RRL.

Formally, the EXPAND operation takes a sparse tensor $\mathcal{F}$ of arity r and creates a new sparse tensor $\mathcal{F}'$ with arity $r + 1$. This is implemented by duplicating each entry $f(x_1, \cdots, x_r)$ in $\mathcal{F}$ by N times, creating the N new vector representations for $(x_1, \cdots, x_r, o_i)$ for all $i \in \{1, 2, \cdots, N\}$, where N is the number of nodes in the hypergraph.

The REDUCE operation takes a sparse tensor $\mathcal{F} = (\mathbf{I}, \mathbf{V})$ of arity r and creates a new sparse tensor $\mathcal{F}'$ with arity $r - 1$: it aggregates all information associated with all r-tuples: $(x_1, x_2, \cdots, x_{r-1}, ?)$ with the same $r - 1$ prefix. In SpaLoc, the aggregation function is chosen to be *max*. Thus,

$$f'(x_1, \cdots, x_{r-1}) = \max_{z:(x_1,\cdots,x_{r-1},z)\in\mathbf{I}} f(x_1, \cdots, x_{r-1}, z).$$

The CONCAT operation concatenates the input hyperedge representations along the channel dimension (i.e., the dimension corresponding to different relational features). Specifically, it

first adds missing entries with all-zero values to the input hyperedge representations so that they have exactly the same set of indices $\mathbf{I}$. It then concatenates the $\mathbf{V}$'s of inputs along the channel dimension.

The PERMUTE operation takes a sparse tensor $\mathcal{F}$ of arity r and creates a new sparse tensor $\mathcal{F}'$ of the same arity. However, the length of the vector representation will grow from D to $D' = r! \times D$. It fuses the representation of hyperedges that share the same set of entities. Mathematically,

$$f'(x_1, \cdots, x_r) = \underset{(x'_1, \cdots, x'_r) \text{ is a permutation of } (x_1, \cdots, x_r)}{\text{CONCAT}} \left[f(x'_1, \cdots, x'_r) \right].$$

If a permutation of $(x_1, \cdots, x_r)$ does not exist in $\mathcal{F}$, it will be treated as an all-zero vector. Thus, the number of entries M may increase or remain unchanged.

Finally, the i-th sparse relational reasoning layer has R+1 linear layers $L^{(i,0)}, L^{(i,1)}, \cdots, L^{(i,R)}$ with nonlinear activations (e.g., ReLU) as submodules with arities 0 through R. For each arity r, we will concatenate the feature tensors expanded from arity $r-1$, those reduced from arity $r+1$, and the output from the previous layer, apply a permutation, and apply $L^{(i,r)}$ on the derived tensor.

To make the intermediate features $\mathcal{F}^{(i,r)}$ sparse, SpaLoc uses a gating mechanism. In SpaLoc, for each linear layer $L^{(i,r)}$, we add a linear gating layer, $L_g^{(i,r)}$, which has sigmoid activation and outputs a scalar value in range $[0, 1]$ that can be interpreted as the importance score for each hyperedge. During training, we modulate the output of $L^{(i,r)}$ with this importance value. Specifically, the output of layer i arity r is $\mathcal{F}^{(i,r)} = L^{(i,r)}(\mathcal{F}) \odot L_g^{(i,r)}(\mathcal{F})$, where $\mathcal{F}$ is the input sparse tensor, and $\odot$ is the element-wise multiplication operation. Note that we are using the same gate value to modulate each channel dimension of $L^{(i,r)}(\mathcal{F})$. During inference, we can prune out edges with small importance scores $L_g^{(i,r)} < \epsilon$, where ϵ is a scalar hyperparameter. We use $\epsilon = 0.05$ in our experiments.

We have described the computation of a sparsified Neural Logic Machine. However, we do not know a *priori* the sparse structures of intermediate layer outputs at training time, nor at inference time before we actually compute the output. Thus, we have to start from the assumption of a fully-connected dense graph. In the following sections, we will show how to impose regularization to encourage learning sparse features. Furthermore, we will present a subsampling technique to learn efficiently from large input graphs.

Remark. Even when the inputs have only unary and binary relations, allowing intermediate tensor representations of higher arity to be associated with hyperedges increases the expressiveness of NLMs. In Chapter 4, we will prove that NLMs with max arity $k+1$ are as expressive as k-GNN hypergraph models (note that the regular GNN is 1-GNN). An

intuitive example is that, in order to determine the *grandparent* relationship, we need to consider all 3-tuples of entities, even though the input relations are only binary. Despite their expressiveness, hyperedge-based NLMs cannot be directly applied to large-scale graphs. For a graph with more than 10,000 nodes, such as Freebase [Bollacker *et al.*, 2008], it is almost impossible to store vector representations for all of the N^3 tuples of arity 3. Our key observation to improve the efficiency of NLMs is that relational rules are usually applied *sparsely* (Section 2.4.2) and *locally* (Section 2.4.3).

2.4.2 Sparsification through Hoyer Regularization

We use a regularization loss to encourage hyperedge sparsity, which is based on the Hoyer sparsity measure (2004). Let x (in our case, the edge gate $g(x_1, \cdots, x_r)$) be a vector of length n. Then

$$Hoyer(x) = \frac{(\sum_i^n |x_i|)/\sqrt{\sum_i^n x_i^2} - 1}{\sqrt{n} - 1}.$$

The Hoyer measure takes values from 0 to 1. The larger the Hoyer measure of a tensor, the denser the tensor is. In order to assign weights to different tensors based on their size, we use the Hoyer-Square measure [Yang *et al.*, 2020],

$$H_S(x) = \frac{(\sum_i^n |x_i|)^2}{\sum_i^n x_i^2},$$

which ranges from 1 (sparsest) to n (densest). Intuitively, the Hoyer-Square measure is more suitable than L_1 or L_2 regularizers for graph sparsification since it encourages large values to be close to 1 and others to be zero, i.e., extremity. It has been widely used in sparse neural network training and has shown better performance than other sparse measures [Hurley and Rickard, 2009]. We empirically compare H_S with other sparsity measures in Appendix B.4.

The overall training objective of SpaLoc is the task objective plus the sparsification loss, $\mathcal{L} = \mathcal{L}_{task} + \lambda \mathcal{L}_{density}$, where $\mathcal{L}_{density}$ is the sum of the H_S, divided by the sum of the sizes of these tensors.

2.4.3 Subgraph Training

Regularization enables us to learn a sparse model that will be efficient at *inference* time, but does not address the problem of *training* on large graphs. We describe a novel strategy that substantially reduces training complexity. It is based on the observation that an inferred relation among a set of entities generally depends only on a small set of other entities that are "related to" the target entities in the hypergraph, in the sense that they are connected via short paths of relevant relations.

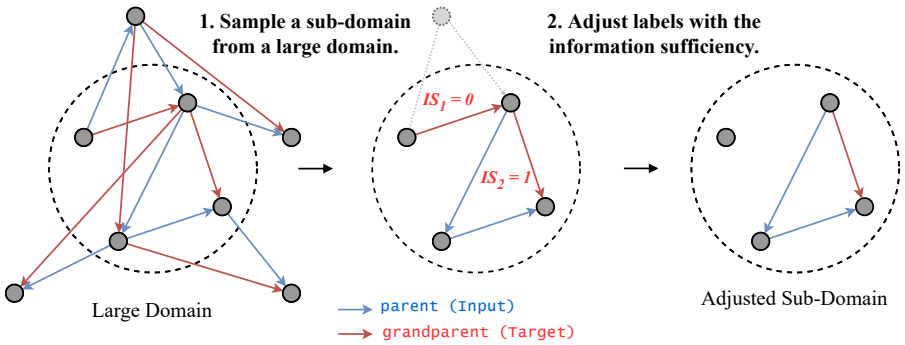


Figure 2-6: **Subgraph sampling and label calibration in SpaLoc.** Subgraph training contains two steps. First, we sample a subset of nodes from the whole graph. Next, we adjust labels for edges in the sub-sampled graph. $IS_1 = 0$ because no paths connecting two nodes are sampled, while $IS_2 = 1$ because all paths connecting two nodes are sampled.

Specifically, we employ a sub-graph sampling and label adjustment procedure. Here, we first present a measure to quantify the sufficiency of information in a sub-sampled graph for determining the relationship between two entities, namely, *information sufficiency*. Next, we present a sub-graph sampling procedure designed to maximize the information sufficiency for training. We further show that sub-graph sampling can also be employed at inference time. Finally, since information loss is inevitable during sampling, we further propose a training label adjustment process based on the information sufficiency.

Information sufficiency. Let $\mathcal{H}_S = (\mathcal{V}_S, \mathcal{E}_S)$ be a sub-hypergraph of hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, and $e^* = (y_1, \ldots, y_r)$ be a target hyperedge in $\mathcal{H}_S$, where $y_1, \cdots, y_r \in \mathcal{V}_S \subset \mathcal{V}$. Intuitively, in order to determine the label for this hyperedge, we need to consider all "paths" that connect the nodes $\{y_1, \ldots, y_r\}$. More formally, we say a sequence of K hyperedges $(e_1, \ldots, e_K)$, represented as

$$\underbrace{(x_1^1, \cdots, x_{r_1}^1)}_{e_1}, \underbrace{(x_1^2, \cdots, x_{r_2}^2)}_{e_2}, \cdots, \underbrace{(x_1^K, \cdots, x_{r_k}^K)}_{e_K},$$

is a *hyperpath* for nodes $\{y_1, \cdots y_r\}$ if and only if $\{y_1, \cdots y_r\} \subset \bigcup_{j=1}^K e_j$ and $e_j \cap e_{j+1} \neq \emptyset$ for all j. In a graph with only binary edges, this is equivalent to the existence of a path from one node y_1 to another node y_2. We define the *information sufficiency* measure for a hyperedge e^* in subgraph $\mathcal{H}_S$ as ($\frac{0}{0}$ is defined as 1.)

$$IS\left((y_1, \cdots, y_r) \mid \mathcal{H}_S, \mathcal{H}\right) := \frac{\#\text{Paths connecting } (y_1, \cdots, y_r) \text{ in } \mathcal{H}_S}{\#\text{Paths connecting } (y_1, \cdots, y_r) \text{ in } \mathcal{H}}.$$

In practice, we approximate IS by only counting the number of paths whose length is less than a task-dependent threshold τ for efficiency. The number of paths in a large graph can be

pre-computed and cached before training, and the overhead of counting paths in a sampled graph is small, so this computation does not add much overhead to training and inference. When input graphs have maximum arity 2, paths can be counted efficiently by taking powers of the graph adjacency matrix.

Subgraph sampling. During training, each data point is a tuple $(\mathcal{H}, f, f')$ where $\mathcal{H}$ is the input graph, f is the input representation, and f' is the desired output labels. We sample a subgraph $\mathcal{H}' \subset \mathcal{H}$, and train models to predict the value of f' on $\mathcal{H}'$ given f. For example, we train models to predict the *grandparent* relationship between all pairs of entities in $\mathcal{H}'$ based on the *parent* relationship between entities in $\mathcal{H}'$. Thus, our goal is to find a subgraph that retains most of the paths connecting nodes in this subgraph. We achieve this using a *neighbor expansion sampler* that uniformly samples a few nodes from $\mathcal{V}$ as the seed nodes. It then samples new nodes connected with one of the nodes in the graph into the sampled graph and runs this "expansion" procedure for multiple iterations to get $\mathcal{V}_S$. Finally, we include all edges that connect nodes in $\mathcal{V}_S$ to form the final subsampled hypergraph.

When the task is to infer the relations between a single pair of entities $f'(y_1, y_2)$ given the input representation f, a similar sub-sampling idea can also be used at inference time to further speed it up. Specifically, we use a *path sampler*, which samples paths connecting y_1 and y_2 and induces a subgraph from these paths. We provide ablation studies on different sampling strategies in Section 2.4.4.

Training label adjustment with IS. Due to the information loss caused by graph subsampling, the information contained in the subgraph may not be sufficient to make predictions about a target relationship. For example, in a family relationship graph, removing a subset of nodes may cause the system to be unable to conclude whether a specific person x has a sibling.

Thus, we propose to adjust the model training by assigning each example $f'(y_1, \cdots, y_r)$ with a soft label, as illustrated in Fig. 2-6. Consider a binary classification task f'. That is, function f' is a mapping from a hyperedge tuple of arity r to $\{0, 1\}$. Denote the model prediction as $\hat{f}'$. Typically, we train the SpaLoc model with a binary cross-entropy loss between $\hat{f}'$ and the ground truth f'. In our subgraph training, we instead compute a binary cross-entropy loss between $\hat{f}'$ and $f'_{\mathcal{H}_S} \odot IS$, where $\mathcal{H}_S$ is the sub-sampled graph. Mathematically,

$$\left(f'_{\mathcal{H}_S} \odot IS\right)(y_1, \cdots, y_r) \triangleq f'_{\mathcal{H}_S}(y_1, \cdots, y_r) \cdot IS\left((y_1, \cdots, y_r) \mid \mathcal{H}_S, \mathcal{H}\right).$$

We empirically compare IS with other label smoothing methods in Appendix B.6.

Next, we compare SpaLoc with other methods in two aspects: accuracy and efficiency

Family Tree	MemNN		∂ILP		NLM		GraIL (R-GCN)		**SpaLoc (Ours)**	
N_{train}	20	2,000	20	2,000	20	2,000	20	2,000	20	2,000
`HasFather`	65.24	-	100	-	100	-	100	100	$100_{\pm 0.00}$	$100_{\pm 0.00}$
`HasSister`	66.21	-	100	-	100	-	97.05	97.95	$100_{\pm 0.00}$	$98.01_{\pm 0.04}$
`Grandparent`	64.57	-	100	-	100	-	99.95	98.08	$100_{\pm 0.00}$	$100_{\pm 0.00}$
`Uncle`	64.82	-	100	-	100	-	97.87	96.50	$100_{\pm 0.00}$	$100_{\pm 0.00}$
`MGUncle`	80.93	-	100	-	100	-	54.67	71.29	$100_{\pm 0.00}$	$100_{\pm 0.00}$
`FamilyOfThree`	-	-	-	-	100	-	-	-	$100_{\pm 0.00}$	$100_{\pm 0.00}$
`ThreeGenerations`	-	-	-	-	100	-	-	-	$100_{\pm 0.00}$	$100_{\pm 0.00}$

Table 2.3: **Results (per-class accuracy) on family tree reasoning benchmarks.** Models are trained on domains with 20 to 2000 entities, and tested on domains with 100 entities. Minus mark means the model runs out of memory or cannot handle ternary predicates. All experiments are conducted on a single NVIDIA 3090 GPU with 24GB memory. The scores and standard errors of SpaLoc are obtained from experiments with three different random seeds.

on large domains. We first compare SpaLoc with other baseline models on a synthetic family tree reasoning benchmark. Since we know the underlying relational rules of the task and have fine-grained control over training/testing distributions, we use this benchmark for ablation studies about the space and time complexity of our model and two design choices (different sampling techniques and different label adjustment techniques). We further extend the results to several real-world knowledge-graph reasoning benchmarks.

2.4.4 Family Tree Reasoning

We first evaluate SpaLoc on a synthetic family-tree reasoning benchmark for inductive logic programming. The goal is to induce target family relationships or member properties in the test domains based on four input relations: *Son*, *Daughter*, *Father*, and *Mother*. Details are defined in Appendix B.8.

Baseline. We compare SpaLoc against four baselines. The first three are Memory Networks [MemNNs; Sukhbaatar *et al.*, 2015], ∂ILP [Evans and Grefenstette, 2018], and Neural Logic Machines. For these models, we follow the configuration and setup in Section 2.3. The fourth baseline is an inductive link prediction method based on graph neural networks, GraIL [Teru *et al.*, 2020]. Since GraIL can only be used for link prediction, we use the full-batch R-GCN [Schlichtkrull *et al.*, 2018], the backbone network of GraIL, for node property predictions.

Accuracy & Scalability. Table 2.3 summarizes the result. Overall, SpaLoc achieves near-perfect performance across all prediction tasks, on par with the inductive logic programming-

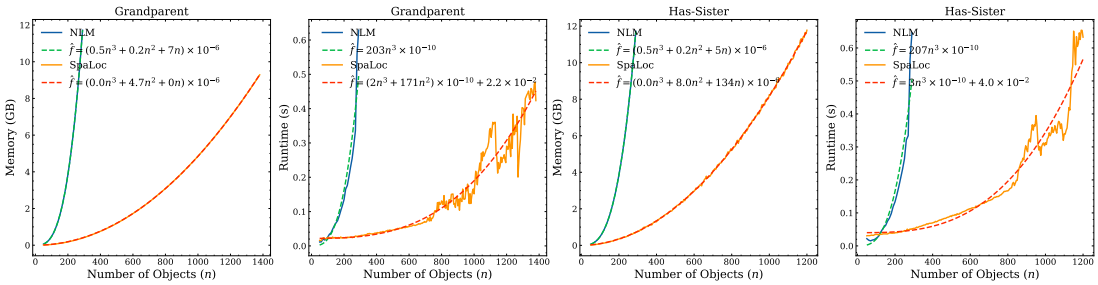


Figure 2-7: **Memory usage and inference time of SpaLoc.** The memory usage and the inference time of each sample *vs*. the number of objects in the evaluation domains. SpaLoc reduces the memory complexity from $O(n^3)$ to $O(n^2)$ and achieves significant runtime speedup.

	Uncle			Grandparent		
	Acc.	Time	Mem.	Acc.	Time	Mem.
NLM	100	133.8	3,846	100	135.0	3,846
SpaLoc + NLM	100	**37.2**	**214**	100	**23.9**	**181**
2-GNN	100	145.1	5,126	100	145.5	5,126
SpaLoc + 2-GNN	100	**23.7**	**645**	100	**19.2**	**519**

Table 2.4: **Performance and runtime complexity of SpaLoc.** Per-class Accuracy, per-sample inference time (ms), and memory usage (MB) when applying SpaLoc on 2-GNNs. Recall that 1-GNN is the standard GNN with only binary edge message passing. Models are tested on domains with 200 entities.

N_s/N	*Node*		*Walk*		*Neighbor*	
	Acc	MIS	Acc	MIS	Acc	MIS
20 / 50	100	54.82	100	85.14	100	89.78
20 / 200	100	33.05	100	71.51	100	80.60
20 / 500	58.18	27.27	100	78.22	100	78.70
20 / 1,000	1.84	24.49	100	77.18	100	78.38
20 / 2,000	0	19.66	100	79.69	100	78.53

Table 2.5: **Comparison of different subgraph samplers in SpaLoc.** The first column shows the size of the sub-sampled graph during training (N_s) and the full training graph (N). Models are tested on domains with 100 entities.

based method ∂ILP and the baseline model NLM. This suggests that our sparsity regularizations and sub-graph sampling do not affect model accuracy. Importantly, our SpaLoc framework has drastically increased the scalability of the method: SpaLoc can be trained on graphs with 2000 nodes, which is infeasible for the baseline NLM model due to memory issues.

Another essential comparison is between GraIL and SpaLoc. GraIL is a graph neural network–based approach that only considers relationships between binary pairs of entities. This is sufficient for simple tasks such as *HasFather*, but not for more complex tasks such as *Maternal Great Uncle (MGUncle)*. By contrast, SpaLoc explicitly reasons about hyperedges and solves more complex tasks.

Runtime & Memory. We study the time and memory complexity of SpaLoc against NLM on the *HasSister* and *Grandparent* tasks. Results are shown in Fig. 2-7, where we plot the curve of average memory consumption and inference time as a function of the input graph size. We fit a cubic polynomial equation to the data points to approximate the learned inference complexity of SpaLoc. The experimental results show that our method can empirically reduce the space complexity from the original $O(n^3)$ complexity of NLM to approximately $O(n^2)$. Note that this learned network has the same complexity as the optimal relational rule that can be designed to solve both tasks. The inference time also gets significantly improved.

Application to other hypergraph neural networks. SpaLoc is a general framework for scaling up hypergraph neural networks rather than a method that can only be used on NLMs. Here we apply our framework SpaLoc to a new method, k-GNN [Morris *et al.*, 2019] on the family tree benchmark. Specifically, we use a fully-connected k-hypergraph. The edge embeddings are initialized as a one-hot encoding of the input relationship. Shown in Table 2.4, we see consistent improvements in terms of inference speed and memory cost for k-GNNs and NLMs.

Ablation: Subgraph sampling. We compare our *neighbor expansion sampler* with two other sub-graph samplers, proposed in Zeng *et al.* [2020b]: random node (*Node*) and random walk (*Walk*) samplers. We compare these samplers with two metrics: the final accuracy of the model and the average information sufficiency of all pairs of nodes in the sub-sampled graphs (MIS). Table 2.5 shows the result on the *Grandparent* task. The *Node* sampler does not leverage locality, so the performance of models and MIS drop as the domain size grows larger. The SpaLocs trained with *Walk* and *Neighbor* samplers perform similarly well in terms of test accuracy. Note that the accuracy results are consistent with the MIS results:

Model	WN18RR				FB15k-237				NELL-995			
	v1	v2	v3	v4	v1	v2	v3	v4	v1	v2	v3	v4
Neural-LP	86.02	83.78	62.90	82.06	69.64	76.55	73.95	75.74	64.66	83.61	87.58	85.69
DRUM	86.02	84.05	63.20	82.06	69.71	76.44	74.03	76.20	59.86	83.99	87.71	85.94
RuleN	90.26	89.01	76.46	85.75	75.24	88.70	91.24	91.79	84.99	88.40	87.20	80.52
GraIL	94.32	94.18	85.80	92.72	84.69	90.57	91.68	94.46	86.05	92.62	93.34	87.50
TACT	96.15	97.95	90.58	96.15	88.73	94.20	97.10	98.30	94.87	96.58	95.70	96.12
SpaLoc	**98.45**	**99.96**	**96.63**	**99.34**	**99.74**	**99.34**	**99.55**	**99.29**	**100**	**98.56**	**97.19**	**97.34**

Table 2.6: **Results (per-class accuracy) on real-world knowledge graph inductive reasoning datasets from GraIL.** The scores and standard errors (appendix Table B.2) of SpaLoc are obtained from experiments with three different random seeds.

comparing the *Node* sampler and others, we see that a higher MIS score leads to higher test accuracy. This supports the effectiveness of our proposed information sufficiency measure.

2.4.5 Real-World Knowledge Graph Reasoning

To further demonstrate the scalability of SpaLoc, we apply it to complete real knowledge graphs. We test SpaLoc on both inductive and transductive relation prediction tasks, following GraIL [Schlichtkrull *et al.*, 2018]. In this setting, the test-task time is to infer the relationship on a given edge, so test-time graph subsampling is used. We evaluate the models with a classification metric, the area under the precision-recall curve (AUC-PR).

In the inductive setting, the training and evaluation graphs are disjoint sub-graphs extracted from WN18RR [Dettmers *et al.*, 2018], FB15k-237 [Toutanova *et al.*, 2015], and NELL-995 [Xiong *et al.*, 2017]. For each knowledge graph, there are four versions with increasing sizes. In the transductive setting, we use the standard WN18RR, FB15k-237, and NELL-995 benchmarks. For WN18RR and FB15k-237, we use the original splits; for NELL-995, we use the split provided in GraIL. We also include the Hit@10 metric used by knowledge graph embedding methods. Following the setting of GraIL, we rank each test triplet among 50 randomly sampled negative triplets.

Baseline. We compare SpaLoc with several state-of-the-art models, including Neural LP [Yang *et al.*, 2017a], DRUM [Sadeghian *et al.*, 2019], RuleN [Meilicke *et al.*, 2018], GraIL, and TACT [Chen *et al.*, 2021a]. For transductive learning tasks, we compare SpaLoc with four representative knowledge graph embedding methods: TransE [Bordes *et al.*, 2013], DistMult [Yang *et al.*, 2015a], ComplEx [Trouillon *et al.*, 2017], and RotatE [Sun *et al.*, 2019b].

	WN18RR		NELL-995		FB15K-237	
	AUC-PR	H@10	AUC-PR	H@10	AUC-PR	H@10
TransE	93.73	88.74	98.73	98.50	98.54	98.87
DistMult	93.08	85.35	97.73	95.68	97.63	98.67
ComplEx	92.45	83.98	97.66	95.43	97.99	**98.88**
RotatE	93.55	88.85	98.54	98.09	98.53	98.81
GraIL	90.91	73.12	97.79	94.54	92.06	75.87
SpaLoc	**96.74**	**99.94**	**99.25**	**98.93**	**99.69**	96.96

Table 2.7: **Results (AUC-PR) on transductive link prediction on real-world knowledge graphs.** We also include Hit@10 as an additional metric following knowledge graph embedding literature and GraIL [Schlichtkrull *et al.*, 2018]. The scores and standard errors (appendix Table B.3) of SpaLoc are obtained from experiments with three different random seeds.

Results. Table 2.6 and Table 2.7 show the inductive and transductive relation prediction results respectively. In the inductive setting, SpaLoc significantly outperforms all baselines on all datasets. This demonstrates the scalability of SpaLoc on large-scale real-world data. SpaLoc is the only model that explicitly uses hyperedge representations, while none of the existing hypergraph neural networks are directly applicable to such large graphs due to memory and time complexities. In the transductive setting, SpaLoc outperforms all knowledge embedding (KE) methods and GraIL on WN18RR and NELL-995. SpaLoc also has comparable performance with KE methods on the FB15K-237 datasets, outperforming GraIL by a large margin. Comparing SpaLoc with node embedding–based methods (TransE) and GNN-based methods (GraIL) that only consider binary edges, we see that our hyperedge-based model enables better relation prediction that requires reasoning about other entities. The necessity of hyperedges is further supported by Appendix B.5, where we show that setting the maximum arity of SpaLoc to 2 (i.e., removing hyperedges) significantly degrades the performance.

Notably, in contrast to other methods for the transductive setting that store entity embeddings for all knowledge graph nodes, SpaLoc directly uses the inductive learning setting. That is, SpaLoc does not store entity information about each knowledge graph node. SpaLoc performs pretty well in the classification setting and ranking setting when negative candidate sets are small (Table 2.7). However, we find inductive reasoning methods, including SpaLoc, GraIL [Teru *et al.*, 2020], and TACT [Chen *et al.*, 2021a], perform poorly in the traditional knowledge graph ranking setting [Bordes *et al.*, 2013] (i.e., ranking the positive triplets against negative triplets of all replacement of head/tail) entities. The ranking scores are close to zero when the negative candidates set are large. We attribute this phenomenon to the fact that the inductive models do not store entity information. We leave better adaptations of

SpaLoc to transductive learning settings as future work.

2.5 Related Work

ILP and relational reasoning. Inductive logic programming (ILP) [Muggleton, 1991, 1996; Friedman *et al.*, 1999] is a paradigm for learning logic rules derived from a limited set of rule templates from examples. Being a powerful way of reasoning over discrete symbols, it is successfully applied to various language-related problems, and has been integrated into modern learning frameworks [Kersting *et al.*, 2000; Richardson and Domingos, 2006; Kimmig *et al.*, 2012]. Recently, Evans and Grefenstette [2018] introduces a differentiable implementation of ILP which works with connectionist models such as CNNs. Sharing a similar spirit, Rocktäschel and Riedel [2017] introduces an end-to-end differentiable logic proving system for knowledge base (KB) reasoning. A major challenge of these approaches is to scale up to a large number of complex rules. Searching a rule as complex as our `ShouldMove` example in Appendix A.5 from scratch is beyond the scope of most systems that use weighted symbolic rules generated from templates.

As shown in Section 2.2.4, both computational complexity and parameter size of the NLM grow *polynomially* w.r.t. the number of allowed predicates (in contrast to the *exponential* dependence in ∂ILP [Evans and Grefenstette, 2018]), but factorially w.r.t. the breadth (max arity, same as ∂ILP). Therefore, our method can deal with more complex tasks such as the blocks world which requires using a large number of intermediate predicates, while ∂ILP fails to search in such a large space.

The method proposed in this chapter also differs from existing approaches on using neural networks to augment symbolic rule induction [Lippi and Frasconi, 2009; Manhaeve *et al.*, 2018]. Specifically, we have *no* rule designed by humans as the input or the knowledge base for the model. NLMs are general neural architectures for learning lifted rules from only input-output pairs.

Our work is also related to symbolic relational reasoning, which has a wide application in processing discrete data structures such as knowledge graphs and social graphs [Zhu *et al.*, 2014; Kipf and Welling, 2017; Zeng *et al.*, 2017; Yang *et al.*, 2017a]. Most symbolic relational reasoning approaches [e.g., Yang *et al.*, 2017b; Rocktäschel and Riedel, 2017] are developed for KB reasoning, in which the predicates on both sides of a rule is known in the KB. Otherwise, the complexity grows exponentially in the number of used rules for a conclusion, which is the case in the blocks world. Moreover, Yang *et al.* [2017a] considers rues of the form $\texttt{query}(\texttt{Y}, \texttt{X}) \leftarrow \texttt{R}_n(\texttt{Y}, \texttt{Z}_n) \wedge \cdots \wedge \texttt{R}_1(\texttt{Z}_1, \texttt{X})$, which is not for general reasoning. The key of Rocktäschel and Riedel [2017] and Campero *et al.* [2018] is to learn subsymbolic embeddings of entities and predicates for efficient KB completion, which differs from our

focus. While NLMs can scale up to complex rules, the number of objects/entities or relations should be bounded as a small value (e.g., < 1000), since all predicates are represented as tensors. This is, to some extent, in contrast with the systems developed for knowledge base reasoning. We leave the scalability of NLMs to large entity sets as future works.

Besides, modular networks [Andreas *et al.*, 2016b, 2017; Mascharka *et al.*, 2018] are proposed for the reasoning over subsymbolic data such as images and natural language question answering. Santoro *et al.* [2017] implements a visual reasoning system based on "virtual" objects brought by receptive fields in CNNs. Wu *et al.* [2017] tackles the problem of deriving structured representation from raw pixel-level inputs. Dai *et al.* [2018] combines structured visual representation and theorem proving.

Graph neural networks and relational inductive bias. Graph convolution networks (GCNs) [Bruna *et al.*, 2014; Li *et al.*, 2016; Defferrard *et al.*, 2016; Kipf and Welling, 2017] is a family of neural architectures working on graphs. As a representative, Gilmer *et al.* [2017] proposes a message passing modeling for unifying various graph neural networks and graph convolution networks. GCNs achieved great success in tasks with intrinsic relational structures. However, most of the GCNs operate on pre-defined graphs with only nodes and binary connections. This restricts the expressive power of models in general-purpose reasoning tasks [Li *et al.*, 2016].

In contrast, this work removes such restrictions and introduces a neural architecture to capture lifted rules defined on any set of objects. Quantitative results support the effectiveness of the proposed model in a broad set of tasks ranging from relational reasoning to modeling general algorithms (as decision-making process). Moreover, being fully differentiable, NLMs can be plugged into existing convolutional or recurrent neural architectures for logic reasoning.

Relational decision making. Logic-driven decision making is also related to Relational RL [Van Otterlo, 2009], which models the environment as a collection of objects and their relations. State transition and policies are both defined over objects and their interactions. Examples include OO-MDP [Diuk *et al.*, 2008; Kansky *et al.*, 2017], symbolic models for learning in interactive domains [Pasula *et al.*, 2007], structured task definition by object-oriented instructions [Denil *et al.*, 2017], and structured policy learning [Garnelo *et al.*, 2016]. General planning methods solve these tasks via planning based on rules [Hu and De Giacomo, 2011; Srivastava *et al.*, 2011; Jiménez *et al.*, 2019]. The goal of this chapter is to introduce a neural architecture which learns lifted rules and handle relational data with multiple orders. We leave its application in other RL and planning tasks as future work.

Neural abstraction machines and program induction. Neural Turing Machine (NTM) [Graves *et al.*, 2014, 2016] enables general-purpose neural problem solving such as sorting by introducing an external memory that mimics the execution of Turing Machine. Neural program induction and synthesis [Neelakantan *et al.*, 2016; Reed and De Freitas, 2016; Kaiser and Sutskever, 2016; Parisotto *et al.*, 2017; Devlin *et al.*, 2017; Bunel *et al.*, 2018; Sun *et al.*, 2018] are recently introduced to solve problems by synthesizing computer programs with neural augmentations. Some works tackle the issue of the systematical generalization by introducing extra supervision [Cai *et al.*, 2017]. In Chen *et al.* [2018], more complex programs such as language parsing are studied. However, the neural programming and program induction approaches are usually hard to optimize in an end-to-end manner, and often require strong supervisions (such as ground-truth programs).

Efficient training and inference methods. There is a rich literature on efficient training and inference of neural networks. Two directions that are relevant to us are model sparsification and sampling training. Han *et al.* [2016] proposed to prune and compress the weights of neural networks for efficiency, and Yang *et al.* [2020] adopted Hoyer-Square regularization to sparsify models. SpaLoc extends this sparsification idea by adding regularization at intermediate sparse tensor groundings to encourage sparse induction. Chiang *et al.* [2019] and Zeng *et al.* [2020b] proposed to sample sub-graphs for GNN training and Teru *et al.* [2020] proposed to construct sub-graphs for link prediction. Cheng *et al.* [2021a] uses pre-defined Horn rules to sample subgraphs to train knowledge graph embeddings. SpaLoc generalizes these sampling methods to hypergraphs and proposes the information sufficiency-based adjustment method to remedy the information loss introduced by sub-sampling.

2.6 Discussion

We present neural logic machines, a neuro-symbolic architecture for first-order logic reasoning with deep neural networks, and SpaLoc, a framework for efficient training and inference of NLMs.

NLMs exploit the power of both neural networks, as function approximators, and logic programming, as a symbolic processor for objects with properties, relations, logic connectives, and quantifiers. After being trained on small-scale tasks (such as sorting short arrays), NLMs can recover lifted rules and generalize to large-scale tasks (such as sorting longer arrays). Furthermore, SpaLoc leverages sparsity and locality to enable efficient training and inference. By regularizing intermediate representations with a sparsification loss, SpaLoc achieves inference complexities on family tree tasks comparable to algorithms designed by human experts, while also generalizing well to large-scale relational reasoning

benchmarks.

The NLM framework establishes a deep connection between first-order quantification structures in logic and neural network architectures. Since the lifted structure in first-order logic is one of the most fundamental structures in reasoning and planning, it serves as a foundation for differentiable program execution and learning. This foundation will reappear in later chapters, such as Chapter 6 and Chapter 11. Next, in Chapter 3, we will extend this formalism to modeling temporal relations, and in Chapter 4, we will discuss the expressiveness of NLMs in detail.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

Neuro-Symbolic Temporal Logic

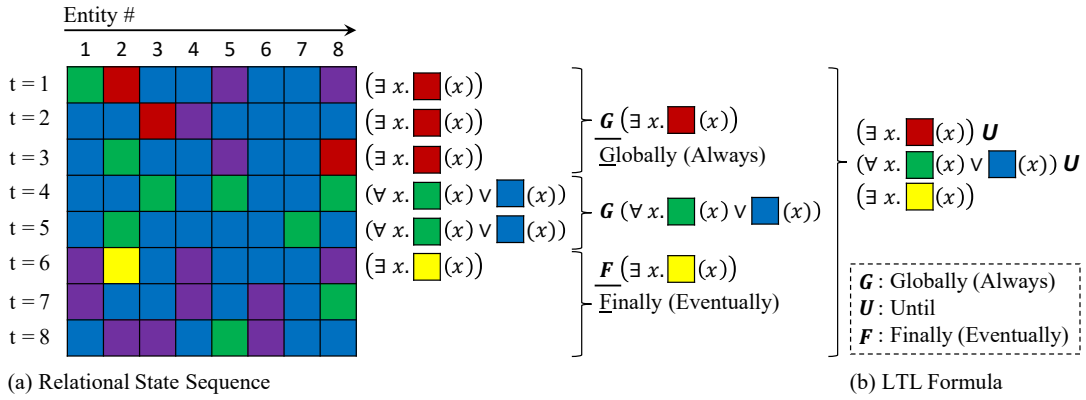
This chapter presents *Temporal and Object Quantification Networks* (TOQ-Nets), a new class of neuro-symbolic networks with a structural bias that enables them to learn to recognize complex relational-temporal events. This is done by including reasoning layers that implement finite-domain quantification over objects and time. The structure allows them to generalize directly to input instances with varying numbers of objects in temporal sequences of varying lengths. We evaluate TOQ-Nets on input domains that require recognizing event-types in terms of complex temporal relational patterns. We demonstrate that TOQ-Nets can generalize from small amounts of data to scenarios containing more objects than were present during training and to temporal warpings of input sequences.

TOQ-Nets can be understood as an extension of the NLM architecture to *linear temporal logic* structures, modeling important temporal relations among events. This extension is important because it paves the way for modeling temporal structures of actions, which we will introduce in Part III.

The material of this chapter has been previously published in Mao *et al.* [2021a]. Zhezheng Luo, in particular, contributed significantly to the materials presented in this chapter.

3.1 Introduction

Every day, people interpret events and actions in terms of concepts, defined over temporally evolving relations among agents and objects [Zacks *et al.*, 2007; Stränger and Hommel, 1996]. For example, in a soccer game, people can easily recognize when one player has control of the ball, when a player *passes* the ball to another player, or when a player is *offsides*. Although it requires reasoning about intricate relationships among sets of objects over time, this cognitive act is effortless, intuitive, and fast. It also generalizes directly over different numbers and arrangements of players, and detailed timings and trajectories.



(a) Relational State Sequence (b) LTL Formula

Figure 3-1: **Describing complex events with FO-LTL formula.** (a) An input sequence composed of relational states: each column represents the state of an entity that changes over time. A logic formula describes a complex concept or feature that is true of this temporal sequence using object and temporal quantification. The sequence is segmented into three stages: throughout the first stage, ■ holds for at least one entity, until the second stage, in which each entity is always either ■ or ■, until the third stage, in which ■ eventually becomes true for at least one of the entities. (b) Such events can be described using first-order linear temporal logic expressions.

In contrast, most computational representations of sequential concepts are based on fixed windows of space and time, and lack the ability to perform relational generalization.

In this chapter, we develop generalizable representations for learning complex activities in time sequences from realistic data. As illustrated in Fig. 3-1, we can describe complex events with a first-order linear temporal logic [FO-LTL; Pnueli, 1977] formula, which allows us to flexibly decompose an input sequence into stages that satisfy different criteria over time. Object quantifiers (∀ and ∃) are used to specify conditions on sets of objects that define each stage. Such representations immediately support generalization to situations with a varying number of objects, and sequences with different time warpings.

More concretely, the variety of complicated spatio-temporal trajectories that *high pass* can refer to in a soccer game can be described in these terms: in a high pass from player A to teammate B, A is *close* to the ball ($\mathit{distance}(A, \mathit{ball}) < \theta_1$) and moving ($\mathit{velocity}(A) > \theta_2$) *until* the ball moves over the ground ($z_{\mathit{position}}(\mathit{ball}) > \theta_3$), which is true *until* teammate B gets control of the ball ($\mathit{teammate}(A, B) \wedge \mathit{distance}(B, \mathit{ball}) < \theta_1$). Beyond modeling human actions in physical environments, these structures can be applied to events in any time sequence of relational states, e.g., characterizing particular offensive or defensive maneuvers in board games such as chess or in actual conflicts, or detecting a process of money-laundering amidst financial transaction records.

We propose a neuro-symbolic approach to learning to recognize temporal relational patterns, called *Temporal and Object Quantification Networks* (TOQ-Nets), in which we design

structured neural networks with an explicit bias that represents finite-domain quantification over both entities and time. A TOQ-Net is a multi-layer neural network whose inputs are the properties of agents and objects and their relationships, which may change over time. Each layer in the TOQ-Net performs either *object* or *temporal* quantification.

Similar to Chapter 2, the key idea of TOQ-Nets is to use *tensors* to represent relations between objects, and to use tensor pooling operations over different dimensions to realize temporal and object quantifiers ($\boldsymbol{G}$ and $\forall$). For example, the colorful matrix in Fig. 3-1(a) can be understood as representing a unary color property of a set of 8 entities over a sequence of 8 time steps. Representing a sequence of relations among objects over time would require a 3-dimensional tensor. Crucially, the design of TOQ-Nets allows *the same network weights* to be applied to domains with different numbers of objects and time sequences of different lengths. By stacking object and temporal quantification operations, TOQ-Nets can easily learn to represent higher-level sequential concepts based on the relations between entities over time, starting from low-level sensory input and supervised with only high-level class labels.

There are traditional symbolic learning or logic synthesis methods that construct first-order or linear temporal logic expressions from accurate symbolic data [Neider and Gavran, 2018; Camacho *et al.*, 2018; Chou *et al.*, 2020]. TOQ-Nets take a different approach and can learn from noisy data by backpropagating gradients, which allows them to start with a general perceptual processing layer that is directly fed into logical layers for further processing.

We evaluate TOQ-Nets on two perceptually and conceptually different benchmarks: trajectory-based sport event detection and human activity recognition, demonstrating several important contributions. First, TOQ-Nets outperform both convolutional and recurrent baselines for modeling temporal-relational concepts across benchmarks. Second, by exploiting temporal-relational features learned through supervised learning, TOQ-Nets achieve strong few-shot generalization to novel actions. Finally, TOQ-Nets exhibit strong generalization to scenarios with more entities than were present during training and are robust w.r.t. time warped input trajectories. These results illustrate the power of combining symbolic representational structures with learned continuous-parameter representations to achieve robust, generalizable interpretation of complex relational-temporal events.

3.2 TOQ-Nets

The input to a TOQ-Net is a tensor representation of the properties of all entities at each moment in time. For example, in a soccer game, the input encodes the position of each player and the ball, as well as their team membership at each step of an extended time

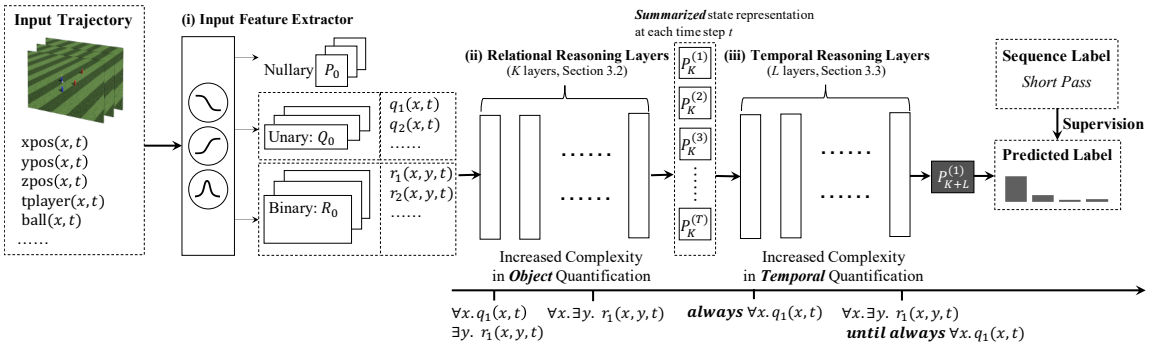


Figure 3-2: **Overview of TOQ-Nets.** A TOQ-Net contains three modules: (i) an input feature extractor, (ii) relational reasoning layers, and (iii) temporal reasoning layers. To illustrate the model's representational power, we show that logical forms of increasing complexity can be realized by stacking multiple layers.

interval. The output is a label of the category of the sequence, such as the type of soccer play it contains.

The first layer of a TOQ-Net (Fig. 3-2 (i)) extracts temporal features for each entity with an *input feature extractor* that focuses on entity features within a fixed and local time window. These features may be computed, e.g., by a convolutional neural network or a bank of parametric feature templates. The output of this step is a collection of nullary, unary, and binary relational features over time for all entities. Throughout the chapter we will assume that all output tensors of this layer are binary-valued, but it can be extended directly to real-valued functions. This input feature extractor is task-specific and is not the focus of this chapter.

Second, these temporal-relational features go through several *relational reasoning* layers (RRLs), detailed in Section 3.2.2, each of which performs linear transformations, sigmoid activation, and *object quantification* operations. The linear and sigmoid functions allow the network to realize learned Boolean logical functions, and the object quantification operators can realize quantifiers. Additional RRLs enable deeper nesting of quantified expressions, as illustrated in Fig. 3-2. All operations in these layers are performed for all time steps in parallel.

Next, the RRLs perform a final quantification, computing for each time step a set of a nullary features that are passed to the *temporal reasoning layers* (TRLs), as detailed in Section 3.2.3. Each TRL performs linear transformations, sigmoid activation, and *temporal quantification*, allowing the model to realize a subset of linear temporal logic [Pnueli, 1977]. As with RRLs, adding more TRLs enables the network to realize logical forms with more deeply nested temporal quantifiers.

In the last layer, all object and time information is projected into a set of features of the initial time step, which summarize the temporal-relational properties of the entire trajectory

(e.g., "the kicker eventually scores"), and fed into a final softmax unit to obtain classification probabilities for the sequence.

It is important to understand the representational power of this model. The *input transformation layer* learns basic predicates and relations that will be useful for defining more complex concepts, but no specific predicates or relations are built into the network in advance. The relational reasoning layers build quantified expressions over these basic properties and relations, and might construct expressions that could be interpretable as "the player is close to the ball." Finally, the temporal reasoning layer applies temporal operations to these complex expressions, such as "the player is close to the ball until the ball moves with high speed." Critically, *none* of the symbolic properties or predicates are hand defined—they are all constructed by the initial layer in order to enable the network to express the concept it is being trained on.

TOQ-Nets are not fully first-order: all quantifiers operate only over the finite domain of the input instance, and can be seen as "short-hand" for finite disjunctions or conjunctions over objects or time points. In addition, the depth of the logical forms it can learn is determined by the fixed depth of the network. However, our goal is not to fully replicate temporal logic, but to bring ideas of object and temporal quantification into neural networks, and to use them as structural inductive biases to build models that generalize better from small amounts of data to situations with varying numbers of objects and time courses.

3.2.1 Temporal-Relational Feature Representation

TOQ-Nets use tensors as internal representations between layers; they represent, all at once, the values of all predicates and relations grounded on all objects at all time points. The operations in a TOQ-Net are vectorized, operating in parallel on all objects and times, sometimes expanding the dimensionality via outer products, and then re-projecting into smaller dimensions via max-pooling. This processing style is analogous to representing an entire graph using an adjacency matrix and using matrix operations to compute properties of the nodes or of the entire graph. In TOQ-Nets, the input to the network, as well as the feature output of intermediate layers, is represented as a tuple of three tensors.

Specifically, we use a vector of dimension D_0 to represent aspects of the state that are global and do not depend on any specific object at each time t. We use a matrix of shape $N \times D_1$ to represent the unary properties of each entity at time t, where N is the number of entities and D_1 is the hidden dimension size. Similarly, we use a tensor of shape $N \times N \times D_2$ to represent the relations between each pair of entities at time step t. As a concrete example, illustrated in Fig. 3-2, the number of entities N is the total number of players plus one (the ball). For each entity x and each time step, the inputs are their 3D position, type (ball or player) and team membership. The TOQ-Net outputs the action performed by the target

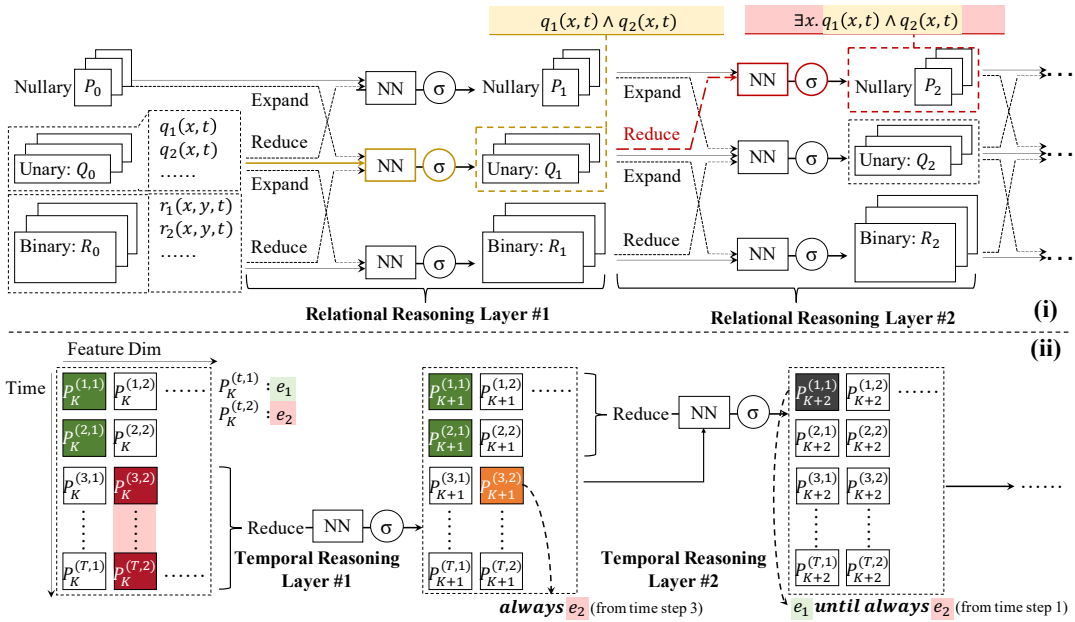


Figure 3-3: **Illustration of relational reasoning layers and temporal reasoning layers.** We provide two illustrative running traces. (i) The first relational reasoning layer takes unary predicates q_1 and q_2 as input and its output Q_1 is able to represent $q_1 \wedge q_2$. The $\max(Q_1, dim = 0)$ in layer 2 can represent $\exists x.\, q_1(x,t) \wedge q_2(x,t)$. (ii) Assume P_K encodes the occurance of events e_1 and e_2 at each time step. The first temporal reasoning layer can realize *always* e_2 with a temporal pooling from time step 3 to time step T. In the second temporal reasoning layer, the temporal pooling summarizes that e_1 holds true from time step 1 to 2. Thus, the NN should be able to realize e_1 *until* (*always* e_2).

player. Since there are only entity features, the input trajectory is encoded with a "unary" tensor of shape $T \times N \times D_1$, where T is the length of the trajectory. That is, there are no nullary or binary inputs in this case.

3.2.2 Relational Reasoning Layers

Our relational reasoning layers (RRLs) follow prior work on Neural Logic Machines presented in Chapter 2, illustrated in Fig. 3-3 (i). Consider a specific time step t. At each layer l, the input to a neural logic layer is a 3-tuple $(P_{l-1}, Q_{l-1}, R_{l-1})$, which corresponds to nullary, unary, and binary features respectively. Their shapes are D_0, $N \times D_1$, and $N \times N \times D_2$. The output is another 3-tuple (P_l, Q_l, R_l), given by

$$
\begin{aligned}
P_l &= \mathrm{NN}_P\left(\mathrm{Concat}\left[P_{l-1}; \max(Q_{l-1}, \mathrm{dim} = 0)\right]\right), \\
Q_l &= \mathrm{NN}_Q\left(\mathrm{Concat}\left[Q_{l-1}; \max(R_{l-1}, \mathrm{dim} = 0);\right.\right. \\
&\qquad \left.\left.\max(R_{l-1}, \mathrm{dim} = 1); \mathrm{expand}(P_{l-1}, \mathrm{dim} = 1)\right]\right), \\
R_l &= \mathrm{NN}_R\left(\mathrm{Concat}\left[R_{l-1}; \mathrm{expand}(Q_{l-1}, \mathrm{dim} = 0);\right.\right. \\
&\qquad \left.\left.\mathrm{expand}(Q_{l-1}, \mathrm{dim} = 1)\right]\right).
\end{aligned}
$$

where NN_* are single fully-connected layers with sigmoid activations. For unary and binary features, NN_Q and NN_R are applied along the feature dimension. That is, we apply the same linear transformation to the unary features of all entities. A different linear transformation is applied to the binary features of all entity pairs. Concat$[\cdot\ ;\ \cdot]$ is the concatenation operation, applied to the last dimension of the tensors (the feature dimension). max, also called the "reduced" max operation, takes the maximum value along the given axis of a tensor. The expand operation, also called "broadcast," will duplicate the input tensor N times and stack them together along the given axis. RRLs are applied identically to the input features at every time step t. That is, we use the same neural network weights in a RRL for all time steps in parallel.

RRLs are motivated by relational logic rules in a finite and fully grounded universe. The max reduction operations implement a differentiable version of an existential quantifier over the finite universe of individuals, given that the truth values of the propositions are represented as values in $(0.0, 1.0)$. Because preceding and subsequent RRLs can negate propositions as needed, we omit explicit implementation of finite-domain universal quantification, although it could be added by including analogous min reductions. Thus, as illustrated in Fig. 3-3 (i), given input features $q_1(x,t)$ and $q_2(x,t)$, we can realize the formula $\exists x.\ q_1(x,t) \wedge q_2(x,t)$ by stacking two such layers.

Throughout the chapter we have been using only nullary, unary, and binary features, but the proposed framework itself can be easily extended to higher-order relational features. From a graph network [Bruna *et al.*, 2014; Kipf and Welling, 2017; Battaglia *et al.*, 2018] point of view, one can treat these features as unary and binary features of a fully-connected graph and the relational reasoning layers as specialized graph neural network layers for realizing object quantifiers.

3.2.3 Temporal Reasoning Layers

Temporal reasoning layers (TRLs) perform *quantification* operations similar to relational reasoning layers, but along the *temporal* rather than the object dimension. The first TRL takes

as input the summarized event representation produced by the K-th relational reasoning layer, $P_K^{(t)}$ for at all time steps t, as a matrix of shape $T \times D$. Each TRL is computed as

$$P_{K+l}^{(t)} = \max_{t'>t} \mathrm{NN}_l\left(\mathrm{Concat}\left[P_{K+l-1}^{(t')}; \max_{t \le t'' < t'} P_{K+l-1}^{(t'')}\right]\right). \tag{3.1}$$

We will walk through the subexpressions of this formula.

1. P_{K+l-1} is the output tensor of the previous temporal reasoning layer, of shape $T \times C$, where T is the number of time steps, and C is the number of feature channels. Each entry in this tensor $P_{K+l-1}^{(t)}[i]$ can be interpreted as: event i happens at time t.

2. $\left(\max_{t \le t'' < t'} P_{K+l-1}^{(t'')}\right)$, abbreviated as $Q_{k+l-1}^{(t,t')}$ in the following text, is a vector of shape C. Its entry $Q_{k+l-1}^{(t,t')}[i]$, where $t \le t'$, represents the concept that event i happens some time between t and t'. **Importantly,** together with the preceding and subsequent neural network operations, which can realize negation operations, it also allows us to describe the event that i holds true for all time steps between t and t'.

3. NN_l is a fully connected neural network with sigmoidal activation, which gets uniformly applied to all time steps t and a future time step $t' > t$. Its input is composed of two parts: the events that happen at t', i.e., $P_{K+l-1}^{(t)}$, and the events that happen between t and t', summarized with temporal quantification operations, i.e., $\left(\max_{t \le t'' < t'} P_{K+l-1}^{(t'')}\right)$.

4. The outer-most max pooling operation $\max_{t'>t}$ enumerates over all $t' > t$, and tests whether the condition specified by NN_l holds for at least one such t'.

A special case is the first temporal reasoning layer. It takes P_K as its input, which is the output of last relational reasoning layer. Thus, the first temporal reasoning layer implements:

$$P_{K+1}^{(t)} = \mathrm{NN}_1\left(\max_{t \le t'' < T} P_K^{(t'')}\right),$$

where T is the sequence length. Note that there is no enumeration for a future time step $t' > t$ involved. In addition, for all temporal layers, we add residual connections by concatenating their inputs with the outputs.

Next, let's consider a running example illustrating how a TOQ-Net can recognize the event that: event p holds true until event q becomes always true. In LTL, this can be written as $p\boldsymbol{U}(\boldsymbol{G}q)$. Using the plain first-order logic (FOL) language, we can describe it as: $\exists\, t.\, [\forall t'.\, (0 \le t' < t) \implies p(t')) \wedge (\forall t'.\, (t' \ge t) \implies q(t')]$.

For simplicity, we consider a tensor representation for two events $p(t)$ and $q(t)$, where $p(t) = 1$ if it happens at time step t and $p(t) = 0$ otherwise. Given the input sequence of length 4 in Table 3.1 ($p(t)$ and $q(t)$), the first layer is capable of computing the following

	Input		1-st Layer				2-nd Layer
t	$p(t)$	$q(t)$	$\boldsymbol{G}p$	$\boldsymbol{F}p$	$\boldsymbol{G}q$	$\boldsymbol{F}q$	$p\ \boldsymbol{U}\ (\boldsymbol{G}q)$
1	T	T	F	T	F	T	T
2	T	F	F	T	F	T	T
3	F	T	F	F	T	T	F
4	F	T	F	F	T	T	F

Table 3.1: **A running example of different temporal quantification formulas that TOQ-Nets can realize.** For clarity, we use T and F for True/False. In the actual computation, they are "soft" Boolean values ranges in $[0, 1]$. **G** means ***always***; **F** means ***Eventually***; **U** means ***until***.

Algorithm 1 An example temporal structure that the second temporal reasoning layer can recognize.

Input: $p(t)$, $q(t)$, $(\boldsymbol{G}p)(t)$, $(\boldsymbol{F}p)(t)$, $(\boldsymbol{G}q)(t)$, and $(\boldsymbol{F}q)(t)$
Output: $(p\boldsymbol{U}(\boldsymbol{G}q)))\,(t)$, which is *true* if p holds *true* from time step t until q becomes always *true*.

```
for t ← 1 to T do
    for t' ← t + 1 to T do
        if ∀t'' ∈ [t, t'). p(t'') and (Gq)(t') then
            (pU(Gq)))(t) ← true
```

four properties for each time step t: $\boldsymbol{G}p$ (***always*** p), which is true if p holds true for all future time steps starting from t, $\boldsymbol{F}p$ (***eventually*** p), which is true if p is true for at least one time step starting from t, and similarly, $\boldsymbol{G}q$ (***always*** q) and $\boldsymbol{F}q$ (***eventually*** q). Overall, together with residual connections, the first layer can recognize six useful events: $p(t)$, $q(t)$ (from residual connection), $\boldsymbol{G}p$, $\boldsymbol{F}p$, $\boldsymbol{G}q$, and $\boldsymbol{F}q$ (by temporal quantification, i.e. pooling operations along the temporal dimension).

The second layer can realize the computation depicted in Algorithm 1. For every time step t, it enumerates all $t' > t$ and computes the output based on

1. the events at t' (represented as $P^{(t')}_{K+l-1}$ in Equation 3.1, concretely the $(\boldsymbol{G}q)(t')$ in the Algorithm 1 example), and

2. the state of another event between t and t' (represented as $\left(\max_{t \le t'' < t'} P^{(t'')}_{K+l-1}\right)$ in Equation 3.1, concretely the $\forall t'' \in [t, t').\ p(t'')$ in the Algorithm 1 example).

From the perspective of First-Order Linear Temporal Logic (FO-LTL), stacking multiple temporal reasoning layers enables us to realize FO-LTL formulas such as:

$$p_1\ \boldsymbol{U}\ p_2\ \boldsymbol{U}\ p_3\ \boldsymbol{U}\ \cdots\ \boldsymbol{U}\ p_k,$$

which is interpreted as p_1 holds true until p_2 becomes true and p_2 holds true from that until p_3 becomes true $\cdots$, and

$$\boldsymbol{F}\, p_1\ \boldsymbol{XF}\ p_2\ \boldsymbol{XF}\ p_3\ \boldsymbol{XF}\ \cdots\ \boldsymbol{XF}\ p_k,$$

which is interpreted as p_1 eventually becomes true and after that p_2 eventually becomes true and after that $\cdots$, and in addition, formulas with interleaved until and eventually quantifiers. Here, ***XF*** is a composition of the ne***X***t operator and the ***F***inally operator in LTL. Meanwhile, as described so far, TOQ-Nets can only nest object quantification inside temporal quantification, so it can represent ***always*** $\exists x.\ q_1(x) \wedge q_2(x)$, but not $\exists x.$ ***always*** $q_1(x) \wedge q_2(x)$. This can be solved by interleaving relational and temporal reasoning layers.

It is important to notice that, by using object and temporal pooling operations together with trainable neural networks to realize logic formulas with object and temporal quantifiers, the idea itself generalizes to a broader set of FO-LTL formulas. We design TOQ-Nets to model only a subset of FO-LTL formulas, because they can be computed efficiently (with only $O(T^2)$ space) and they are expressive enough for the type of data we are trying to model.

3.3 Experiments

We compare our model with other approaches to object-centric temporal event detection and to concept learning over robot object-manipulation trajectories in this section. The setups and metrics focus on data efficiency and generalization.

3.3.1 Baseline Approaches

We compare TOQ-Nets against five baselines. The first two are spatial-temporal graph convolutional neural networks [Yan *et al.*, 2018] and its variant STGCN-MAX, which models entity relationships with graph neural networks and models temporal structure with temporal-domain convolutions. The third is STGCN-LSTM, which uses STGCN layers for entity relations but LSTM [Hochreiter and Schmidhuber, 1997] for temporal structures. The last two baselines are based on space-time graphs: Space-Time Graph [Wang and Gupta, 2018] and Non-Local networks [Wang *et al.*, 2018b]. We provide details about our implementation and how we choose the model configurations in Appendix C.1.

Model	Reg.	Few-Shot	Full
STGCN	$73.2_{\pm 1.6}$	$26.0_{\pm 5.7}$	$62.8_{\pm 0.6}$
STGCN-MAX	$73.6_{\pm 1.5}$	$28.6_{\pm 5.0}$	$63.6_{\pm 0.7}$
STGCN-LSTM	$72.7_{\pm 1.4}$	$23.8_{\pm 5.9}$	$61.9_{\pm 0.6}$
Space-Time	$74.8_{\pm 1.5}$	$31.7_{\pm 6.1}$	$65.2_{\pm 0.6}$
Non-Local	$76.5_{\pm 2.4}$	$45.0_{\pm 6.3}$	$69.5_{\pm 2.4}$
TOQ-Net (ours)	$\mathbf{87.7}_{\pm 1.3}$	$\mathbf{52.2}_{\pm 6.3}$	$\mathbf{79.8}_{\pm 0.8}$

Table 3.2: **Results on the soccer event dataset.** Different columns correspond to different action sets (the regular, few-shot, and full action sets). The performance is measured by per-action (macro) accuracy, averaged over nine few-shot splits. The ± values indicate standard errors. TOQ-Net significantly outperforms all baseline methods on the few-shot action set.

3.3.2 Trajectory-Based Soccer Event Detection

We start our evaluation with an event-detection task in soccer games. The task is to recognize the action performed by a specific player at specific time step in a soccer game trajectory.

Dataset and setup. We collect training and evaluation datasets based on the gfootball simulator*, which provides a physics-based 3D football simulation. It also provides AI agents that can be used to generate random plays. The simulator provides the 3D coordinates of the ball and the players as well as the action each player is performing at each time step. There are in total 13 actions defined in the simulator, including *movement, ball_control, trap, short_pass, long_pass, high_pass, header, shot, deflect, catch, interfere, trip* and *sliding*. We exclude *header* and *catch* actions, as they never appear in AI games. We also exclude *ball_control* and *movement*, since they just mean the agent is moving (with or without the ball). Thus, in total, we have nine action categories. We run the simulator with AI-controlled players to generate plays, and formulate the task as classifying the action (9-way classification) of a specific player at a specific time step given a temporal context (25 frames). For each action, we have generated 5,000 videos, expect for *sliding*, for which we generated 4,000 videos because it is rare in the AI games. Among the generated examples, 62% (2,462 or 3,077) are used for training, 15% are used for validation, and 23% are used for testing. Each trajectory is an 8-fps replay clip that contains 17 frames (about two seconds). There is a single "target" player in each trajectory. The action label of the trajectory is the action performed by this target player at frame #9 in the simulator. We randomly split all actions into two categories: seven "regular" actions, for which all game plays are available, and two "few-shot" actions, for which only 50 clips are available during training.

*https://research-football.dev/

Model	Time Warp	3v3	4v4	6v6	8v8	11v11
STGCN	N	$40.7_{\pm 1.0}$ (-40.4%)	$63.2_{\pm 4.9}$ (-7.4%)	$68.2_{\pm 2.8}$ (0.0%)	$55.4_{\pm 3.3}$ (-18.8%)	$44.4_{\pm 2.1}$ (-34.9%)
STGCN	Y	$32.6_{\pm 2.8}$ (-52.3%)	$50.2_{\pm 4.4}$ (-26.5%)	$52.8_{\pm 7.0}$ (-22.6%)	$43.2_{\pm 4.9}$ (-36.7%)	$34.0_{\pm 3.7}$ (-50.2%)
STGCN-MAX	N	$47.4_{\pm 3.2}$ (-33.7%)	$68.8_{\pm 2.0}$ (-3.8%)	$71.5_{\pm 1.9}$ (0.0%)	$59.1_{\pm 0.7}$ (-17.3%)	$45.6_{\pm 2.5}$ (-36.2%)
STGCN-MAX	Y	$37.5_{\pm 7.2}$ (-47.5%)	$52.3_{\pm 4.2}$ (-26.9%)	$56.5_{\pm 4.5}$ (-21.0%)	$46.6_{\pm 3.7}$ (-34.8%)	$36.9_{\pm 1.9}$ (-48.4%)
STGCN-LSTM	N	$39.7_{\pm 1.1}$ (-43.1%)	$60.4_{\pm 0.2}$ (-13.5%)	$69.8_{\pm 0.1}$ (0.0%)	$55.8_{\pm 2.0}$ (-20.0%)	$44.1_{\pm 0.7}$ (-36.8%)
STGCN-LSTM	Y	$21.8_{\pm 0.8}$ (-68.8%)	$27.8_{\pm 1.3}$ (-60.2%)	$30.6_{\pm 0.6}$ (-56.1%)	$25.8_{\pm 1.0}$ (-63.1%)	$22.6_{\pm 0.8}$ (-67.6%)
Space-Time	N	$29.0_{\pm 1.6}$ (-60.4%)	$53.5_{\pm 3.2}$ (-27.0%)	$73.3_{\pm 0.3}$ (0.0%)	$33.9_{\pm 2.8}$ (-53.7%)	$15.2_{\pm 1.8}$ (-79.3%)
Space-Time	Y	$29.7_{\pm 3.1}$ (-59.5%)	$51.6_{\pm 2.5}$ (-29.6%)	$70.7_{\pm 0.3}$ (-3.5%)	$33.8_{\pm 2.2}$ (-53.9%)	$14.9_{\pm 1.6}$ (-79.7%)
Non-Local	N	$45.9_{\pm 5.1}$ (-41.2%)	$70.7_{\pm 5.3}$ (-9.5%)	$78.1_{\pm 5.8}$ (0.0%)	$58.5_{\pm 10.8}$ (-25.1%)	$41.8_{\pm 13.6}$ (-46.5%)
Non-Local	Y	$46.7_{\pm 4.1}$ (-40.2%)	$69.9_{\pm 4.7}$ (-10.5%)	$77.7_{\pm 5.0}$ (-0.5%)	$58.7_{\pm 12.6}$ (-24.9%)	$41.3_{\pm 13.6}$ (-47.1%)
TOQ-Net	N	$\mathbf{77.4}_{\pm 3.5}$ (-12.4%)	$\mathbf{88.3}_{\pm 0.7}$ (-0.0%)	$\mathbf{88.4}_{\pm 0.6}$ (0.0%)	$81.3_{\pm 1.7}$ (-8.0%)	$\mathbf{77.1}_{\pm 1.7}$ (-12.8%)
TOQ-Net	Y	$76.0_{\pm 2.6}$ (-14.0%)	$87.7_{\pm 1.4}$ (-0.8%)	$\mathbf{86.9}_{\pm 0.4}$ (-1.7%)	$80.3_{\pm 1.1}$ (-9.1%)	$74.9_{\pm 2.3}$ (-15.2%)

Table 3.3: **Generalization results on the soccer event dataset.** Full results on generalization to scenarios with all combinations of #agents and temporally warping on the soccer event dataset. The standard error of all values are smaller than 2.5%, computed based on three random seeds.

Input features. Each trajectory is represented with 7 time-varying unary predicates, including the 3D coordinate of each player and the ball and four extra predicates defining the type of each entity x: IsBall(x), IsTargetPlayer(x), SameTeam(x), OpponentTeam(x), where SameTeam(x) and OpponentTeam(x) indicates whether x is of the same team as the target player. We also add a temporal indicator function which is a Gaussian function centered at frame of interest with variance $\sigma^2 = 25$.

Results. Table 3.2 shows the results. Our model significantly outperforms all the baselines in all three action settings, suggesting that our model is able to discover a set of useful features at both input and intermediate levels and use them to compose new action classifiers from only a few examples.

Generalization to more players. Due to their object-centric design, TOQ-Nets can generalize to soccer games with a varying number of agents. After training on 6v6 soccer games (i.e., 6 players on each team), we evaluate the performance of different models on games with different numbers of players: 3v3, 4v4, 8v8, and 11v11. For each action we have generated, on average, 1,500 examples for testing. Table 3.3 summarizes the result. Comparing the columns highlighted in yellow, we notice a significant performance drop for all baselines while TOQ-Net performs the best. By visualizing data and predictions, we found that misclassifications of instances of *shot* as *short pass* contribute most to the performance degradation of our model when we have more players. Specifically, the recall of *shot* drops from 97% to 60%. In soccer plays with many agents, a shot is usually unsuccessful and a player from another team steals the ball in the end. In such scenarios, TOQ-Net tends to misclassify such trajectories as a *short pass*. Ideally, this issue should be addressed by understanding actions based on agents' goals instead of the actual outcome [Intille and Bobick, 2001]. We leave this extension as a future direction.

Generalization to temporally warped trajectories. Another crucial property of TOQ-Nets is to recognize actions based on their sequential order in the input trajectory, instead of binding features to specific time steps in the trajectory. To show this, we test the performance of different models on time warped trajectories. Each test trajectory has a length of 25, and each trajectory is labeled by the action performed by the target player at any time step between the 6-th and 19-th frame. We ensure that the target player performs only one action during the entire input trajectory. Thus, the label is unambiguous. The results are shown in Table 3.3. Specifically, our test set consists of 25-frame trajectories, and the action may occur at anytime between the 6th and the 19th frame. By comparing rows with and without time warping, we notice a 60% performance drop for STGCN, STGCN-MAX,

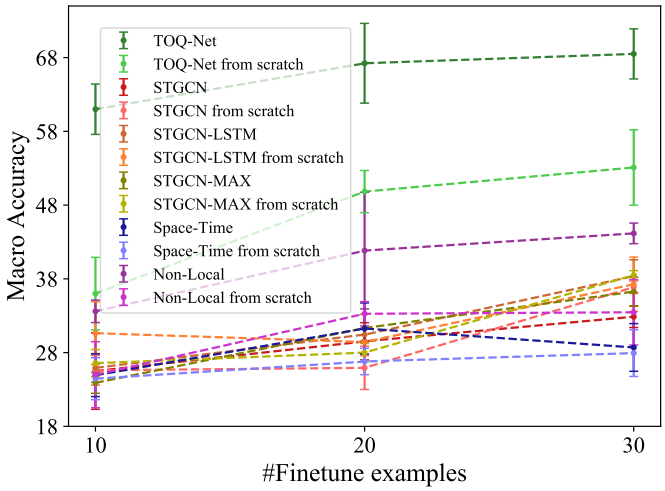


Figure 3-4: **Generalization to soccer environments with a different court size and agent speeds.** The standard errors are computed based on three random seeds.

and STGCN-LSTM. In contrast, TOQ-Nets still have reasonable performance. Note that Space-Time and Non-Local model have almost no performance drop against time warping because they are completely agnostic to temporal ordering.

Generalization to different input scales Intuitively, after learning how to detect events in a soccer game, the learner should also generalize concepts to analogs with the same basic structure but enacted on proportionately larger or smaller spatial or temporal scales—as in the soccer-inspired game futsal. To evaluate generalization to different input scales, we first train all models on the original dataset with 9-way classification supervision. After that, we finetune each model on a new dataset where all input values are doubled (i.e., the court is now double the size, and the players now move at four times the speed), but with only a small number of examples. Thus, time flows twice fast, and the player moves four times the speed compared with the original dataset. Results are summarized in Fig. 3-4. While all baselines can benefit from the pre-training on the original dataset, our TOQ-Net outperforms all baselines by a significant margin when the finetuning dataset is small.

Ablation study: network size. In general, TOQ-Net has a smaller number of weights than baselines. We add additional comparisons to models of different #params. Specifically, model_S is a smaller version of the model and model_T is the tiny version of the model whose #params are about equal to or smaller than #params of TOQ-Net. To obtain smaller models we typically reduce the number of layers and the hidden state dimensions. On the few-shot task of the soccer event dataset, we test all baselines with all combination of features and capacities. Table 3.4 summarizes the results. In general, the "small" variations are the best

Model	#params	Reg.	Few-Shot	Full
STGCN	3.42M	$72.0_{\pm 1.5}$	$22.6_{\pm 5.2}$	$61.0_{\pm 0.8}$
STGCN$_S$	263K	$73.0_{\pm 1.5}$	$22.5_{\pm 4.9}$	$61.7_{\pm 0.5}$
STGCN$_T$	34K	$73.2_{\pm 1.6}$	$26.0_{\pm 5.7}$	$\mathbf{62.8}_{\pm 0.6}$
STGCN-LSTM	2.08M	$72.5_{\pm 1.6}$	$20.2_{\pm 5.0}$	$60.9_{\pm 0.7}$
STGCN-LSTM$_S$	233K	$72.1_{\pm 1.7}$	$20.5_{\pm 5.9}$	$60.6_{\pm 0.7}$
STGCN-LSTM$_T$	33K	$72.7_{\pm 1.4}$	$23.8_{\pm 5.9}$	$\mathbf{61.9}_{\pm 0.6}$
STGCN-MAX	3.42M	$73.5_{\pm 4.8}$	$20.2_{\pm 6.0}$	$61.7_{\pm 4.1}$
STGCN-MAX$_S$	263K	$73.6_{\pm 1.5}$	$28.6_{\pm 5.0}$	$\mathbf{63.6}_{\pm 0.7}$
STGCN-MAX$_T$	34K	$74.5_{\pm 1.2}$	$25.4_{\pm 6.1}$	$63.6_{\pm 0.7}$
Space-Time	263K	$71.3_{\pm 1.4}$	$26.9_{\pm 7.1}$	$61.4_{\pm 1.2}$
Space-Time$_S$	24K	$74.7_{\pm 1.4}$	$30.7_{\pm 6.8}$	$64.9_{\pm 0.6}$
Space-Time$_T$	14K	$74.8_{\pm 1.5}$	$31.7_{\pm 6.1}$	$\mathbf{65.2}_{\pm 0.6}$
Non-Local	1.23M	$74.5_{\pm 4.2}$	$44.3_{\pm 7.0}$	$67.8_{\pm 3.7}$
Non-Local$_S$	108K	$76.5_{\pm 2.4}$	$45.0_{\pm 6.3}$	$\mathbf{69.5}_{\pm 2.4}$
Non-Local$_T$	32K	$75.6_{\pm 1.2}$	$44.8_{\pm 6.4}$	$68.8_{\pm 1.0}$
TOQ-Net	35K	$87.7_{\pm 1.3}$	$52.2_{\pm 6.3}$	$79.8_{\pm 0.8}$

Table 3.4: **Results on the soccer event dataset for baselines with different capacities** Here, model$_S$ and model$_T$ denote the small and the tiny variant for each model. The performance is measured by per-action (macro) accuracy, averaged over nine few-shot splits. The ± values indicate standard errors. For each baseline we showed the best performance over three levels of capacities. In fact, performances of most of the models are not affected much by the capacity. We highlight the best-performing variants of all baselines, and use them in all comparisons in the main text.

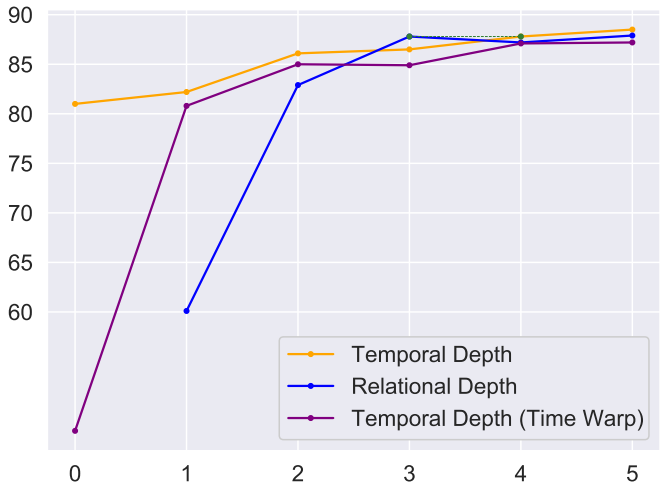


Figure 3-5: **Comparing # of reasoning layers in TOQ-Nets.** Accuracy is tested on 6v6 9-way classification. When # of relational reasoning layers vary (blue), temporal reasoning layers are fixed at 4. When # of temporal reasoning layers vary (orange), relational reasoning layers are fixed at 3. When we have 0 temporal reasoning layers, we predict the sequence label based on the feature of the frame of interest. The purple line shows the performance on temporally warped trajectories when we have the # of relational reasoning layers fixed and vary the # of temporal reasoning layers. Relational layers are required for the prediction task (# of relational reasoning layer must be greater than 0), because there is no nullary feature input to the network: we need at least one relational reasoning layer to gather information across the entities. Overall, adding reasoning layers improves results, and 3 relational layers + 4 temporal layers is a good balance between computation and performance.

for all models (except for STGCN-LSTM). Across all experiments presented in the chapter, we use the best architecture in Table 3.4.

Ablation study: #layers of TOQ-Net. We study the TOQ-Net performance over varying numbers of relation reasoning layers and temporal reasoning layers on the 9-way classification task of the soccer event dataset (See figure Fig. 3-5), and decide to use the combination of 3 relational layers and 4 temporal layers. Both relational layers and temporal layers play important roles in the performance. As a generalization test, we also visualize the accuracy on temporally warped trajectories. We can see that temporal reasoning layers are important for the robustness against temporal warping.

Interpretability of TOQ-Net. Though interpretability is not the main focus of this chapter, in Fig. 3-6, we show how TOQ-Net composes low-level concepts into high-level concepts: from inputs, to intermediate features at different temporal layers, and the final output labels. See the figure caption for detailed analysis.

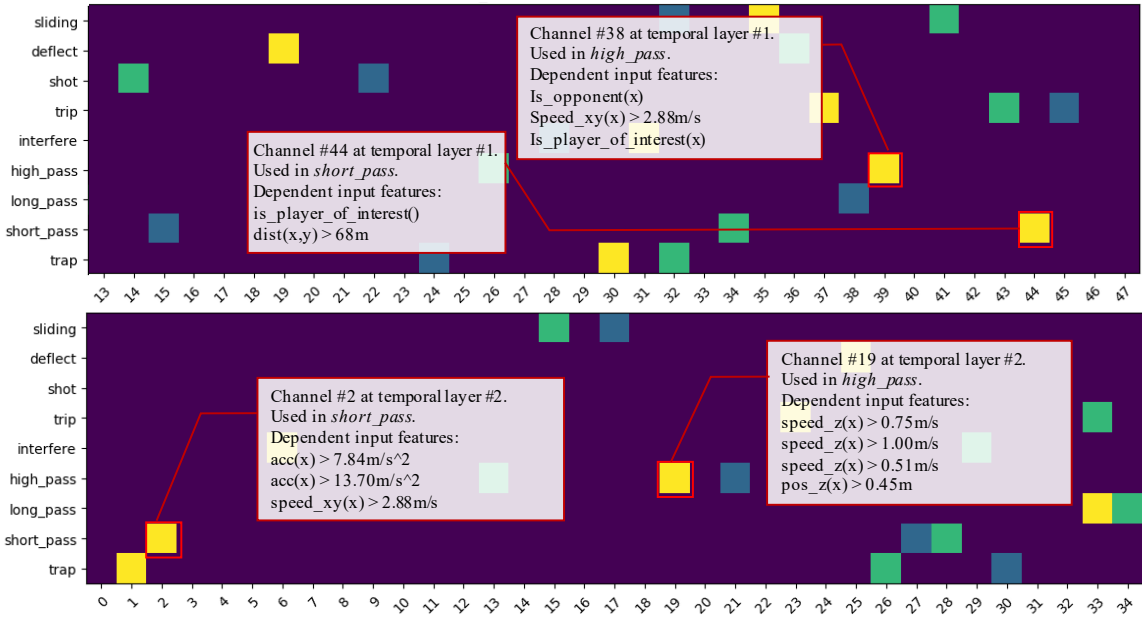


Figure 3-6: **Relevant features in temporal layers in TOQ-Nets.** Feature dependencies are computed by gradient. These dependencies and thresholds are learned end-to-end from data. Insets detail features for events in the first and the second stage of the *high pass* and *short pass*.

3.3.3 Manipulation Concept Learning from 6D Poses

Structural action representations can also be usefully applied to other domains. Here we show the result in a robotic environment, where the goal is to classify the action performed by a robotic arm.

Dataset and setup. We generated a dataset based on the RLBench simulator [James *et al.*, 2020], which contains a set of robotic object-manipulation actions in a tabletop environment. We use 24 actions from the dataset, including *CloseBox, CloseDrawer, CloseFridge, CloseGrill, CloseJar, CloseLaptopLid, CloseMicrowave, GetIceFromFridge, OpenBox, OpenFridge, OpenMicrowave, OpenWineBottle, PickUpCup, PressSwitch, PushButtons, PutGroceriesInCupboard, PutItemInDrawer, PutRubbishInBin, PutTrayInOven, ScoopWithSpatula, SetTheTable, SlideCabinetOpenAndPlaceCups, TakePlateOffColoredDishRack,* and *TakeToiletRollOffStand*. We randomly initialize the position and orientation of different objects in the scene and use the built-in motion planner to generate robot arm trajectories. Depending on the task and the initial poses of different objects, trajectories may have different lengths, varying from 30 to 1,000. Most of the trajectories have ~50 frames.

For each action category, we have generated 100 trajectories. Among the generated examples, 60% of the examples are used for training, 15% are used for validation, and the

other 25% are used for testing.

Input. Each trajectory contains the poses of different objects, at each time step. Each object except for the robot arm is represented as 13 unary properties, including the 3D position, the 3D orientation represented as quaternions (4D vector), and the 3D bounding box (axis-aligned, 6D vector). The robot arm is represented as 8 unary predicates, including the 3D position, the 3D orientation, and a binary predicate that indicates whether the gripper is open or closed. Note that non-rigid objects are split into several pieces so that the learner can infer their state (open or closed). For example, the lid of the box is a separate object. We also input the 6D pose and the state of the robot gripper. During training, we used on average 65 robot trajectories for regular actions.

Input features. All methods take the raw trajectories as input. In TOQ-Nets, we use an extra single linear layer with sigmoid activation as the input feature extractor. To evaluate temporal modeling with recurrent neural networks, STGCN-LSTM uses kernel size 1 in its (STGCN) temporal convolution layers. More ablation studies about baselines can be found in Appendix C.1.

Model	Reg.	1-Shot	Full
STGCN	$99.89_{\pm 0.05}$	$94.92_{\pm 1.03}$	$98.79_{\pm 0.23}$
STGCN-LSTM	$\mathbf{99.92}_{\pm 0.03}$	$95.48_{\pm 1.67}$	$\mathbf{98.86}_{\pm 0.43}$
TOQ-Net	$\mathbf{99.96}_{\pm 0.02}$	$\mathbf{98.04}_{\pm 0.97}$	$\mathbf{99.48}_{\pm 0.24}$

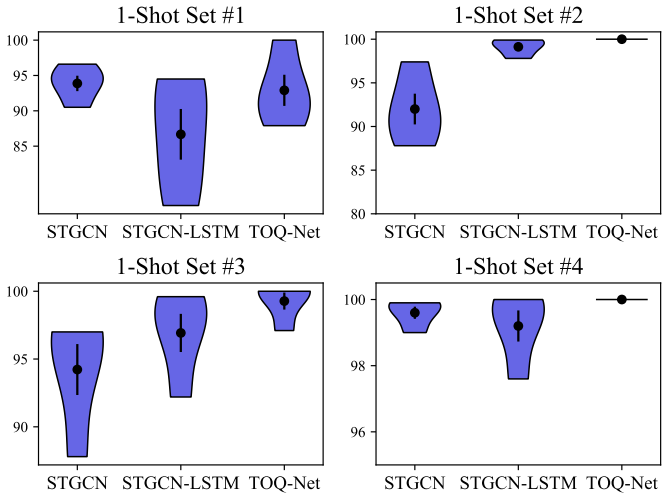
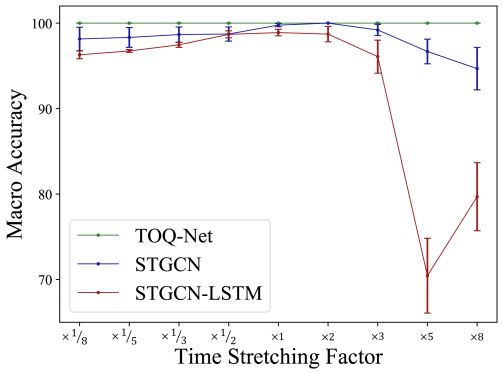


Table 3.5: **Few-shot learning on the RLBench dataset.** Performance is measured by per-action (macro) accuracy and averaged of four 1-shot splits and four random seeds per split. The ± values indicate standard errors. On the right we shows the sampled performance of different models on each individual 1-shot split.



l

Figure 3-7: **Comparing different models with different time stretching factors on the RLBench dataset.**

ni

Results. We start with evaluating different models on the standard 24-way classification task. We summarize results in Fig. 3-7. It also illustrates each model's performance under different time stretching factors of the input trajectory, from $^1/_8\times$ to $8\times$. Similar to the soccer dataset, the TOQ-Net outperforms both convolutional and recurrent models across all time stretching factors.

Generalization to new action concepts. We also evaluate how well different models can generalize the learned knowledge about opening and closing other objects to novel objects. Specifically, we hold out 50% of all *Open X* actions and 50% of the *Close X* actions to form the 1-shot action learning set. For example, the learner may only see a single instance of the opening of a box during training. Actions with more examples are "regular." All models are tested on a 3-way classification task: *Open-X*, *Close-X*, and *Other*. Results are summarized in Table 3.5. There are several held-out sets that are noticeably more "difficult" than others, such as the Set #1 in Table 3.5. Our visualization shows that actions *close jar* and *close drawer* are harder generalization targets, compared to other actions. This is possibly because the motion of *close jar* involves some rotation, and the motion of *close drawer* looks more like pushing than other *close-X* actions.

3.3.4 Extension to Real-World Datasets

The proposed TOQ-Net can also be extended to other real-world datasets. These examples further illustrate the robustness of TOQ-Net to temporal variations in activities.

Toyota Smarthome. Toyota Smarthome [Das *et al.*, 2019] is a dataset that contains videos of humans performing everyday activities such as "walk", "take pills", and "use laptop".

It also comes with 3D-skeleton detections. There are around 16.1k videos in the dataset, and 19 activity classes. The videos' length varies between a few seconds to 3 minutes. We subsample 30 frames for each video. We split frames into training (9.9k), validation (2.5k), and testing (3.6k). We treat human joints as entities. The input is then the position of joints, the velocity of joints, limb lengths, and joint angles. We evaluated our model and STGCN on a 19-way classification task. We also test model performance on time-warped sequences by accelerating the trajectories by two times.

Our model achieves a comparable accuracy to STGCN on the standard classification task: (42.0% vs. 43.0%). Importantly, on the generalization test to time-warped sequences, our model has only a 0.8% performance drop (41.2%), while STGCN drops 10.7% (32.3%). This indicates that the temporal structures learned by TOQ-Net improve model generalization to varying time courses.

Volleyball Activity. The volleyball dataset [Ibrahim *et al.*, 2016] contains 4830 video clips collected from 55 youtube volleyball videos. They are labeled with 8 group activities (e.g. "left spike" and "right pass"). Each video contains 20 frames with the labeled group activity performed at the 10-th frame. The dataset also includes annotations for players, including the bounding box, the indicator of whether the player is involved in the group activity, and the individual action such as "setting", "digging", and "spiking". We use the manual annotations (processed by an MLP) as the input features. We train models to classify the video into one of the eight group activities, following the original split, i.e., 24, 15, and 16 of 55 videos are used for training, validation, and testing.

On the standard classification task, TOQ-Net achieves a comparable performance with STGCN (73.3% vs. 73.6%). When we perform time warping on the input sequences, STGCN's performance drops by more than 25.0% (39.5% on temporally shifted trajectories and 48.6% on 2× quick motion trajectories), while our model drops only 3% (70.3% on temporally shifted trajectories and 70.7% on quick motion trajectories). This again shows the generalization ability of TOQ-Net w.r.t. varying time courses, and the robustness of learned temporal structures.

3.4 Related Work

Action concept representations and learning. First-order and linear temporal logics [LTL; Pnueli, 1977] have been used for analyzing sporting events [Intille and Bobick, 1999, 2001] and activities of daily living [Tran and Davis, 2008; Brendel *et al.*, 2011] in logic-based reasoning frameworks. However, these frameworks require extra knowledge to annotate relationships between low-level, primitive actions and complex ones, or performing

search in a large combinatorial space for candidate temporal logic rules [Penning *et al.*, 2011; Lamb *et al.*, 2007]. By contrast, TOQ-Nets enable end-to-end learning of complex action descriptions from sensory input with only high-level action-class labels.

Temporal and relational reasoning. This chapter is also related to work on using data-driven models for modeling relational and temporal structure, such as LTL [Neider and Gavran, 2018; Camacho *et al.*, 2018; Chou *et al.*, 2020], Logical Neural Networks [Riegel *et al.*, 2020], ADL description languages [Intille and Bobick, 1999], and hidden Markov models [Tang *et al.*, 2012]. These models need human-annotated action descriptions and symbolic state variables (e.g., *pick up x* means a state transition from not *holding x* to *holding x*), and dedicated inference algorithms such as graph structure learning. In contrast, TOQ-Nets have an end-to-end design, and can be integrated with other neural networks. People have also used structural representations to model object-centric temporal concepts with graph neural networks [Yan *et al.*, 2018], recurrent neural networks [RNNs; Ibrahim *et al.*, 2016], and integrated GNN-RNN architectures [Deng *et al.*, 2016; Qi *et al.*, 2018]. TOQ-Nets use a similar relational representation, but different models for temporal structures.

3.5 Discussion

The design of TOQ-Nets suggests multiple research directions. For example, the generalization of the acquired action concepts to novel object kinds, such as from *opening fridges* to *opening bottles*, needs further exploration. Meanwhile, TOQ-Nets are based on physical properties, e.g. 6D poses. Incorporating representations of mental variables such as goals, intentions, and beliefs can aid in action and event recognition [Baker *et al.*, 2017; Zacks *et al.*, 2001; Vallacher and Wegner, 1987].

In summary, we have presented TOQ-Nets, a neuro-symbolic architecture for learning to describe complex state sequences with quantification over both entities and time. TOQ-Nets can be understood as an extension of the NLM architecture to *linear temporal logic* structures, modeling important temporal relations among events. This extension is important because it paves the way for modeling temporal structures of actions, which we will introduce in Part III. Just like NLMs, TOQNets use tensors to represent the time-varying properties and relations of different entities, and use tensor pooling operations over different dimensions to realize temporal and object quantifiers. TOQ-Nets generalize well to scenarios with varying numbers of entities and time courses. Later, in Chapter 11, we will use a temporal structure very similar to the event structure discussed in this chapter to model actions, particularly their preconditions and postconditions. The same idea of differentiable quantification will also carry over.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Expressiveness of Neuro-Symbolic Logic Models

This chapter investigates the use of neural logic machines presented in Chapter 2 and a general class of relational neural networks as goal-conditioned policies. Previously, these policies were generally understood to be "feed-forward" circuits, in the form of neural networks that map from the current state and the goal specification to the next action to take. However, under what circumstances such a policy can be learned and how efficient the policy will be are not well understood. In this chapter, we present a circuit complexity analysis for relational neural networks representing policies for planning problems, by drawing connections with serialized goal regression search (S-GRS). We show that there are three general classes of planning problems, in terms of the growth of circuit width and depth as a function of the number of objects and planning horizon, providing constructive proofs. These proofs are based on a theoretical characterization of the expressiveness of NLMs. We also illustrate the utility of this analysis for designing neural networks for policy learning. These insights further motivate the development of hybrid imperative–declarative representations of actions in Chapter 12.

The material of this chapter has been previously published in Mao *et al.* [2023a] and also includes materials from Luo *et al.* [2021]. Zhezheng Luo, in particular, contributed significantly to the materials presented in this chapter.

4.1 Introduction

Goal-conditioned policies are generally understood to be "feed-forward" circuits, in the form of neural networks such as multi-layer perceptrons (MLPs) or transformers [Vaswani *et al.*, 2017]. They take a representation of the current world state and the goal as input,

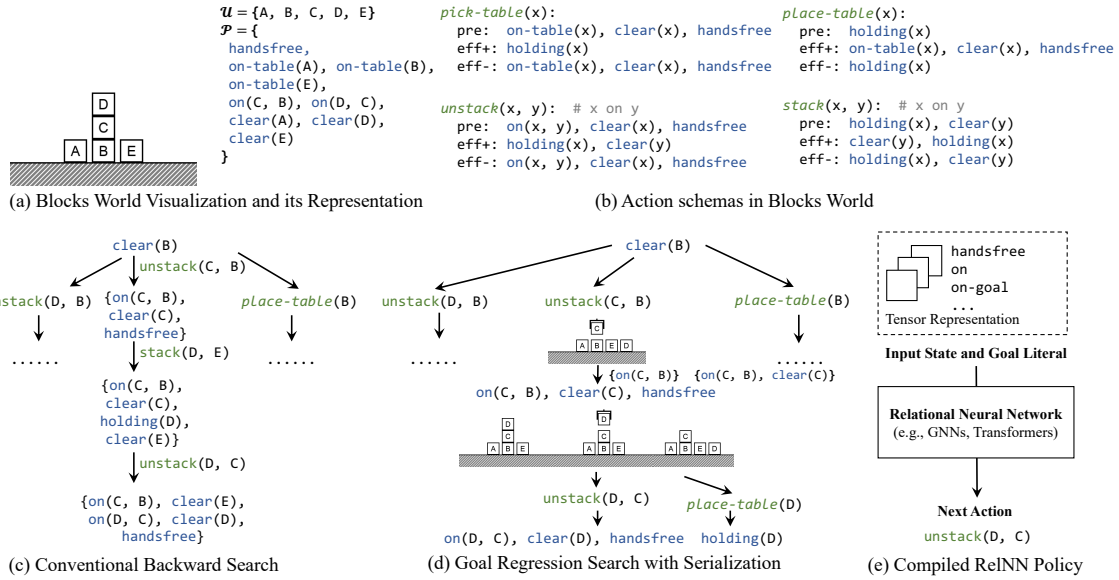
and generate an action as output. Previous work has proposed methods for learning such policies for particular problems via imitation or reinforcement learning [Chapter 2; Wang *et al.*, 2018a; Li *et al.*, 2020c]; recently, others have tried probing current large-language models to understand the extent to which they already embody policies for a wide variety of problems [Liang *et al.*, 2023; Carta *et al.*, 2023].

However, a major theoretical challenge remains. In general, we understand that planning problems are PSPACE-hard with respect to the size of the state space [Bylander, 1994], but there seem to exist efficient (possibly suboptimal) policies for many problems such as block stacking (Chapter 2) that generalize to arbitrarily sized problem instances. In this chapter, we seek to understand and clarify the circuit complexity of goal-conditioned policies for classes of "classical" discrete planning problems: under what circumstances can a polynomial-sized policy circuit be constructed, and what is its size? Specifically, we highlight a relationship between a problem hardness measure (*regression width*, related to the notion of "width" in the forward planning literature [Chen and Giménez, 2007]) and circuit complexity. We concretely provide upper bounds on the policy circuit complexity as a function of the problem's regression width, using constructive proofs that yield algorithms for directly compiling a planning problem description into a goal-conditioned policy. We further show that such policies can be learned with conventional policy gradient methods from environment interactions only, with the theoretical results predicting the necessary size of the network.

There are several useful implications of these results. First, by analyzing concrete planning domains, we show that for many domains, there do exist simple policies that will generalize to problem instances with an arbitrary number of objects. Second, our theory predicts the circuit complexity of policies for these problems. Finally, our analysis suggests insights into why certain planning problems, such as Sokoban and general task and motion planning (TAMP) problems, are hard, and likely cannot be solved *in general* by fixed-sized MLP or transformer-based policies [Culberson, 1997; Vega-Brown and Roy, 2020]. In the rest of the chapter, Section 4.2 provides problem definitions, Sections 4.3 and 4.4 provide complexity definitions, theoretical results, and the policy compilation algorithm; Section 4.5 discusses the practical implications of these results.

4.2 Preliminaries: Planning Domain and Problem

Throughout this chapter, we focus on analyzing the search complexity and policy circuit complexity of classical planning problems. Importantly, these planning problems have an object-centric representation and sparsity in the transition models: the state of the world is represented as a set of entities and their properties and relationships, while each action



(a) Blocks World Visualization and its Representation

(b) Action schemas in Blocks World

(c) Conventional Backward Search

(d) Goal Regression Search with Serialization

(e) Compiled RelNN Policy

Figure 4-1: **Illustration of the Blocks World domain and search trees.** (a) Illustration of the Blocks World domain that we will be using as the main example. (b) The action schema definition in Blocks World. *clear*(x) means there is no object on x. (c) A backward search tree for solving the goal *clear*(B). (d) A serialized goal regression search tree for the same goal. (e) The form of a goal-conditioned policy for this problem.

only changes a sparse set of properties and relations of a few objects. These features will contribute to search efficiency.

Formally, we consider the problem of planning and policy learning for a space $\mathcal{S}$ of world states. A planning problem is a tuple $\langle \mathcal{S}, s_0, \mathcal{G}, \mathcal{A}, \mathcal{T} \rangle$, where $s_0 \in \mathcal{S}$ is the initial state, $\mathcal{G} \subseteq \mathcal{S}$ is a goal specification, $\mathcal{A}$ is a set of actions that the agent can execute, and $\mathcal{T}$ is a (possibly partial) environmental transition model $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. The task of planning is to output a sequence of actions $\bar{a}$ in which the terminal state s_T induced by applying a_i sequentially following $\mathcal{T}$ satisfies $s_T \in \mathcal{G}$. The task of policy learning is to find a function $\pi(s, \mathcal{G})$ that maps from the state and a goal specification to the next action so that applying π recurrently on s_0 eventually yields a state $s_T \in \mathcal{G}$.

We use atomic STRIPS [Fikes and Nilsson, 1971; Lifschitz, 1987] to formalize the object-centric factorization of our planning problems. Specifically, as illustrated in Fig. 4-1a, the environmental state can be represented as a tuple $(\mathcal{U}_s, \mathcal{P}_s)$, where $\mathcal{U}_s$ is the set of objects in state s, denoted by arbitrary unique names (e.g., A, B). The second component, $\mathcal{P}_s$, is a set of *atoms*: *handfree*(), *on-table*(A), *on*(C, B), etc. Each atom contains a predicate name (e.g., *on*) and a list of arguments (e.g., C, B). All atoms in $\mathcal{P}_s$ are true at state s, while any atoms not in $\mathcal{P}_s$ are false. Since we do not consider object creation and deletion, we simply

use $\mathcal{P}_0$ to denote the set of all possible atoms for the object universe $\mathcal{U}_{s_0}$*.

Illustrated in Fig. 4-1b, a (possibly partial) transition model $\mathcal{T}$ is specified in terms of a set of object-parameterized action schemas $\langle name, args, precond, effect \rangle$, where *name* is the name of the action, *args* is a list of symbols, *precond* and *effect* are descriptions of the action's effects. In the basic STRIPS formulation, both *precond* and *effect* are sets of atoms with variables in *args*, and *effect* is further decomposed into eff_+ and eff_-, denoting "add" effects and "delete" effects. An action schema (e.g., *unstack*) can be grounded into a concrete action a (e.g., *unstack*(D, C)) by binding all variables in *args*. The formal semantics of STRIPS actions is: $\forall s. \forall a. pre(a) \subseteq s \implies (\mathcal{T}(s,a) = s \cup eff_+(a) \setminus eff_-(a))$. That is, for any state s and any action a, if the preconditions of a are satisfied in s, the state resulting from applying a to s will be $s \cup eff_+(a) \setminus eff_-(a)$. Note that $\mathcal{T}$ may not be defined for all (s, a) pairs. Furthermore, we will only consider cases where the goal specification is a single atom, although more complicated conjunctive goals or goals that involve existential quantifiers can be easily supported by introducing an additional "goal-achieve" action that lists all atoms in the original goal as its precondition. From now on, we will use g to denote the single goal atom of the problem.

4.3 Goal Regression Search and Width

A simple and effective way to solve STRIPS problems is backward search†. Illustrated in Algorithm 2‡, we start from the goal $\{g\}$, and search for the last action a that can be applied to achieve the goal (i.e., $g \in eff_+(a)$). Then, we add all the preconditions of action a to our search target and recurse. To avoid infinite loops, we also keep track of a "*goal stack*" variable composed of all *goal_set*'s that have appeared in the depth-first search stack. The run-time of this algorithm has critical dependencies on (1) the number of steps in the resulting plan and (2) the number of atoms in the intermediate goal sets in search. In general, it has a worst-case time complexity exponential in the number of atoms.

4.3.1 Serialized Goal Regression Search

One possible way to speed up backward search is to *serialize* the subgoals in the goal set: that is, to commit to achieving them in a particular order. In the following sections, we study a variant of the plain backward search algorithm, namely *serialized goal regression*

*For simplicity, throughout the chapter, we will be focusing on Boolean state variables.

†Fundamentally, backward search has the same worst-case time complexity as forward search. We choose to analyze backward search because goal regression rules are helpful in determining fine-grained search complexity.

‡We show the recursive version for clarity. This algorithm can also be implemented as a breadth-first search.

Algorithm 2 Plain backward.

function BWD(s_0, $\mathcal{A}$, *goal_set*)
  if *goal_set* $\subseteq s_0$ **then return** empty_list()
  possible_t = empty_set()
  for $a \in \{a \in \mathcal{A} \mid$ *goal_set* $\cap$ *eff*$_-(a) = \emptyset$ **and** *goal_set* $\cap$ *eff*$_+ \neq \emptyset\}$ **do**
    new_goals = *goal_set* $\cup$ *pre*$(a) \setminus$ *eff*$_+(a)$
    if *new_goals* $\in$ *goal stack* **then continue**
    π = BWD(s_0, *new_goals*)
    if $\pi \neq \perp$ **then**
      possible_t.add(π + {a})
  return shortest path in *possible_t*

Algorithm 3 Serialized goal regression.

function S-GRS(s_0, $\mathcal{R}$, g, *cons*)
  if $g \in s_0$ **then return** (s_0)
  possible_t = empty_set()
  for $r \in \{r \in \mathcal{R} \mid$ *cons* $\cap$ *eff*$_-$(*action*$(r)) = \emptyset$ **and** *goal*$(r) = g\}$ **do**
    $p_1, p_2, \cdots, p_k$ = subgoals(r)
    if $\exists p_i$ s.t. $p_i \in$ *goal stack* **then continue**
    for i in $1, 2, \cdots, k$ **do**
      new_c = *cons* $\cup \{p_1, \cdots, p_{i-1}\}$
      π_i = S-GRS(s_{i-1}, p_i, *new_c*)
      if $\pi_i == \perp$ **then break**
      $s_i = \mathcal{T}(s_{i-1}, \pi_i[-1])$
    if $\pi_k \neq \perp$ **then**
      possible_t.add($\pi_1 + \ldots + \pi_k$ + {a})
  return shortest path in *possible_t*

search (S-GRS, Algorithm 3), which uses a sequential decomposition of the preconditions: given the goal atom g, it searches for the last action a and also an order in which the preconditions of a should be accomplished. Of course, this method cannot change the worst-case complexity of the problem (actually, it may slow down the algorithm in cases where we need to search through many orders of the preconditions), but there are useful subclasses of planning problems for which it can achieve a substantial speed-up.

A *serialized goal regression rule* in $\mathcal{R}$, informally, takes the following form: $g \leftsquigarrow p_1, p_2, \cdots, p_k \parallel a$. It reads: in order to achieve g, which is a single atom, we need to achieve atoms $p_1, \cdots, p_k$ sequentially, and then execute action a. To formally define the notion of achieving atoms $p_1, \cdots, p_k$ "*sequentially*," we must include constraints in goal regression rules. Formally, we consider the application of a rule under constraint c to have the following definition, $g^c \leftsquigarrow p_1^c, p_2^{c\cup\{p_1\}}, \cdots, p_k^{c\cup\{p_1,\cdots p_{k-1}\}} \parallel a$ which reads: in order to achieve g without deleting atoms in set c, we can first achieve p_1 while maintaining c, then achieve p_2 while maintaining c and p_1, ..., until we have achieved p_k. Finally, we perform a to achieve g. For example, in Blocks World, we have the rule: $clear(\mathrm{B})^{\emptyset} \leftsquigarrow on(\mathrm{A},\mathrm{B}), clear(\mathrm{A})^{\{on(\mathrm{A},\mathrm{B})\}}, handsfree()^{\{on(\mathrm{A},\mathrm{B}),clear(\mathrm{A})\}} \parallel unstack(\mathrm{A},\mathrm{B})$. Note that many rules might be infeasible for a given state, because it is simply not possible to achieve p_i while maintaining c and $\{p_1, \cdots, p_{i-1}\}$. In this case, achieving *handsfree*() before *clear*(A) is infeasible because *handsfree*() may be falsified while achieving *clear*(A), if we need to move other blocks that are currently on A. Similarly, $on(\mathrm{A},\mathrm{B})$ must be achieved before *clear*(A) because moving A onto B will break *clear*(A) (because we need to pick up A).

The set of all possible goal regression rules can be instantiated based on all ground actions $\mathcal{A}$ in a planning problem. Specifically, for a given atom g (e.g., *holding*(A)) and a set of constraints c, and for each action $a \in \mathcal{A}$ (e.g., *pick-table*(A)), if $g \in \mathit{eff}_+(a)$ while $c \cap \mathit{eff}_-(a) = \emptyset$, then for any permutation of $\mathit{pre}(a)$, there will be a goal regression rule: $g^c \leftsquigarrow p_1, \cdots, p_k \parallel a$, where $p_1, \cdots, p_k$ is a permutation of $\mathit{pre}(a)$. We call this regression rule set $\mathcal{R}_0$.

Given a set of regression rules (e.g., $\mathcal{R}_0$), we can apply the serialized goal-regression search (S-GRS) algorithm, shown in Algorithm 3. S-GRS returns a shortest path from *state* to achieve *goal* while maintaining *cons*. Unfortunately, this algorithm is not optimal nor even complete (e.g., a sequence of preconditions may not be "serializable," which we will define formally later), in the general case. To make it complete, it is necessary to backtrack through different action sequences to achieve each subgoal, which increases time complexity. We include a discussion in Appendix D.1.1.

4.3.2 Serialization of Goal Regression Rules

Although Algorithm 3 is not complete in general, it provides insights about goal regression. In the following, we will introduce two properties of planning problems such that S-GRS becomes optimal, complete, and efficient. For brevity, we define $\mathit{OptSearch}(\mathit{state}, \mathit{goal}, \mathit{cons})$ as the set of optimal trajectories that achieve *goal*, from *state*, while maintaining *cons*. Here, *goal* can be a conjunction.

Definition 4.3.1 (Optimal serializability)**.** *A goal regression rule $g^{cons} \leftsquigarrow p_1, \cdots, p_k \parallel a$ is optimally serializable w.r.t. a state s if and only if, for all steps i, if $\forall \pi_{<i} \in \mathit{OptSearch}(s, p_1 \wedge \cdots \wedge p_{i-1}, cons)$ and $\forall \pi_i \in \mathit{OptSearch}(\mathcal{T}(s, \pi_{<i}), p_i, cons \cup \{p_1, \cdots, p_{i-1}\})$ then $\pi_{<i} \oplus \pi_i \in \mathit{OptSearch}(s, p_1 \wedge \cdots \wedge p_i, cons)$. Furthermore, $\forall \pi \in \mathit{OptSearch}(s, p_1 \wedge \cdots \wedge p_k, cons)$, $\pi \oplus a \in \mathit{OptSearch}(s, g, cons)$*[§].

Intuitively, a rule is optimally serializable if any optimal plan for the length $i-1$ prefix of its preconditions can be extended into an optimal plan for achieving the length i prefix. For example, in Blocks World, the rule *holding*(A) $\leftsquigarrow$ *on*(A, B), *clear*(A), *handsfree*() $\parallel$ *unstack*(A, B) is optimally serializable when *on*(A, B) is true for s. We define the set of optimally serializable rules for a state s as $\mathit{OSR}(s)$, and the set of single-literal goals that can be solved with $\mathit{OSR}(\cdot)$ from s_0 as $\mathit{OSG}(s_0)$[¶].

Theorem 4.3.1. *For any goal $g \in \mathit{OSG}(s_0)$, S-GRS is optimal and complete. See proof in Appendix D.1.2.*

[§] $a \oplus b$ denotes concatenation of sequences. $\pi_{<i}$ denotes the subsequence $\{a_0, \cdots, a_{i-1}\}$

[¶] It is possible to make a non-optimally-serializable rule into an optimally serializable one by introducing "super predicates" as the conjunction of existing predicates, at the cost of increasing the state and action space sizes. We include discussions in Appendix D.1.3.

4.3.3 Width of Search Problems

Chen and Giménez [2007] introduced the notion of the *width* of a planning problem and showed that the forward-search complexity is exponential in the width of a problem. We extend this notion to regression-based searches and introduce a generalized version of regression rules in which not all previously achieved preconditions must be explicitly maintained.

Definition 4.3.2 (Generalized regression rules)**.** *A generalized, optimally-serializable regression rule is* $g^c \rightsquigarrow p_1^{c\cup c_1}, p_2^{c\cup c_2}, \cdots, p_k^{c\cup c_k} \parallel a$*, where* $c_i \subseteq \{p_1, p_2, \cdots, p_{i-1}\}$*. Its width is* $|c| + \max\{|c_1|, \cdots, |c_k|\}$.

For example, for any initial state s_0 that satisfies $on(\mathrm{A}, \mathrm{B})$, the following generalized rule is optimally serializable: $clear(\mathrm{B})^{\emptyset} \rightsquigarrow on(\mathrm{A}, \mathrm{B})^{\emptyset}, clear(\mathrm{A})^{\emptyset}, handsfree()^{\{clear(\mathrm{A})\}} \parallel unstack(\mathrm{A}, \mathrm{B})$. When we plan for the second precondition $clear(\mathrm{A})$, we can ignore the first condition $on(\mathrm{A}, \mathrm{B})$, because the optimal plan for $clear(\mathrm{A})$ will not change $on(\mathrm{A}, \mathrm{B})$. Using such generalized rules to replace the default rule set $\mathcal{R}_0$ improves search efficiency by reducing the number of possible subgoals. To define existence conditions for highly efficient search algorithms and policies, we need a stronger notion:

Definition 4.3.3 (Strong optimally-serializable (SOS) width of regression rules)**.** *A generalized regression rule has strong optimally-serializable width k w.r.t. state s if (a) it is optimally serializable, (b) its width is k, and, (c)* $\forall c.OptSearch(s, p_i \cup c_i \cup c, \emptyset) \subseteq OptSearch(s, \{p_1, \cdots, p_i\} \cup c, \emptyset)$.

In the Blocks World domain, given a particular state s, if $on(\mathrm{A}, \mathrm{B}) \in s$, the generalized goal regression rule $clear(\mathrm{B}) \rightsquigarrow on(\mathrm{A}, \mathrm{B}), clear(\mathrm{A}), handsfree()^{\{clear(\mathrm{A})\}} \parallel unstack(\mathrm{A}, \mathrm{B})$ is strong optimally-serializable, because the optimal trajectory to achieve $clear(\mathrm{A})$ will not move A.

Definition 4.3.4 (SOS width of problems)**.** *A planning problem* $P = \langle \mathcal{S}, s_0, g, \mathcal{A}, \mathcal{T} \rangle$ *has strong optimally-serializable width k if there exists a set of strong optimally-serializable width k rules* $\mathcal{R}$ *w.r.t.* s_0*, such that S-GRS can solve P using only rules in* $\mathcal{R}$.

Theorem 4.3.2. *If a problem P is of SOS width k, it can be solved in time* $O(N^{k+1})$ *with* S-GRS*, where N is the number of atoms. Here we have omitted polynomials related to enumerating all possible actions and their serializations (which are polynomial w.r.t. the number of objects in* $\mathcal{U}$*).*

Proof idea. We start by assuming knowing $\mathcal{R}$: the proof can be simply done by analyzing function calls: there are at most $O(N^{k+1})$ possible argument combinations of $(goal, cons)$.

Next, since the enumeration of all possible width-k rules can be done in polynomial time, the algorithm runs in polynomial time and it does not need to know $\mathcal{R}$ a priori. Note that the SOS condition cannot be removed because even if regression rules have a small width, there will be a possibly exponential branching factor caused by "free variables" in regression rules that do not appear in the goal atom. See the full proof in Appendix D.1.4. We also describe the connections with classical planning width and other related concepts, as well as how to generalize to $\forall$-quantified preconditions in Appendix D.1.5.

Importantly, planning problems such as Blocks World and Logistics have constant SOS widths, independent of the number of objects. Therefore, there exist polynomial-time algorithms for finding their solutions. Our analysis is inspired by and closely related to classical (forward) planning width Chen and Giménez [2007] and Lipovetzky and Geffner [2012]. Indeed, we have:

Theorem 4.3.3. *Any planning problem that has SOS-width k has a forward width of at most $k+1$ and, hence, can be solved by the algorithm IW$(k+1)$. See the full proof in Appendix D.1.6.*

Remark. Unfortunately, SOS width and forward width are not exactly equivalent to each other. There exist problems whose forward width is smaller than $k+1$, where k is the SOS width. Appendix D.1.6 presents a concrete example. However, there are two advantages of our SOS width analysis over the forward width analysis. In particular, proofs for (forward) widths of planning problems were mostly done by analyzing solution structures. By contrast, our constructions form a new perspective: *a problem has a large width if the number of constraints to track during a goal regression search is large*. This view is helpful because our analyses can now focus on concrete regression rules for each individual operator in the domain — how its preconditions can be serialized. Second and more importantly, our "backward" view allows us to reason about circuit complexities of policies.

4.4 Policy Realization

We have studied sequential backward search algorithms for planning; now, we address the questions of how best to "compile" or "learn" them as feed-forward circuits representing goal-conditioned policies, and how the size of these circuits depends on properties of the problem. Here, we will focus on understanding the complexity of a certain *problem class*; that is, a set of problems in the same domain (e.g., Blocks World) with a similar goal predicate (e.g., having one particular block clear). In particular, we will make use of *relational neural networks* (*RelNN*, such as graph neural networks [Morris *et al.*, 2019], Transformers [Vaswani *et al.*, 2017], and neural logic machines introduced in Chapter 2).

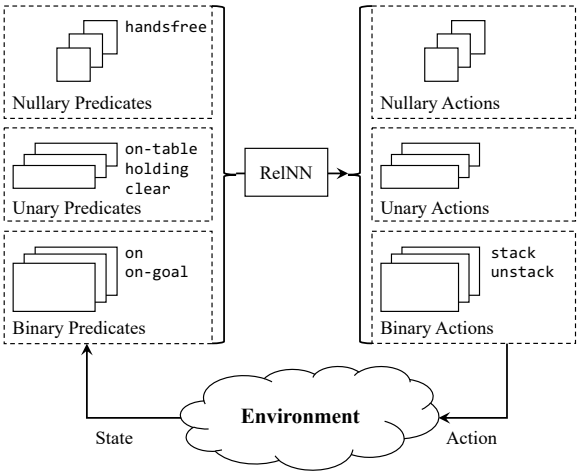


Figure 4-2: **The input and output of a relational neural network (*RelNN*) policy.**

They accept inputs of arbitrary size, which is critical for building solutions to problems with arbitrary-sized universes. Another critical fact is that these networks have "parameter-tying" in the sense that there is a constant-sized set of parameters that is re-used according to a given pattern to realize a network.

In particular, as illustrated in Fig. 4-2, we will show how to take the set of operator descriptions for a planning domain and construct a relational neural network that represents a goal-conditioned policy: it takes a state and a goal (encoded as a special predicate, e.g., *on-goal*) and outputs a ground action. Here, the state is represented as a graph (objects are nodes, and relations are edges; possibly there will be hyperedges for high-arity relations). For example, the input contains nullary features (a single vector for the entire environment), unary features (a vector for each object), binary features (a vector for each pair of objects), etc. The goal predicate can be represented similarly. These inputs are usually represented as tensors. The complexity of this neural network will depend on the SOS width of problems in the problem class of interest.

We will develop two compilation strategies. First, in Section 4.4.2, we directly compile the BWD and the S-GRS algorithms into *RelNN* circuits. For problems with constant SOS width that is independent of the size of the universe, there will exist finite-breadth circuits (but the depth may depend on the size of the universe). In Section 4.4.3, we discuss a new property of planning problems—the existence of *regression rule selector* circuits—which results in policy circuits that are smaller than those generated by previous strategies, which are potentially of finite depth. The construction of the regression rule selector is not automatic and will generally require human priors or machine learning.

4.4.1 Relational Neural Network Expressiveness

We first quantify what a relational neural network can compute. Let depth D be the number of layers in a RelNN, and breadth B be the maximum arity of the relations (hyperedges) in the network. For example, to represent a vector embedding for each tuple of size 5, $f(x_1, x_2, \cdots, x_5)$, where $x_1, \cdots, x_5 \in \mathcal{U}$ are entities in the planning problem, we need a relational neural network with breadth 5. We will only consider networks with a constant breadth. We denote the family of relational neural networks with depth D and breadth B as $RelNN[D, B]$. We will not be modeling the actual "hidden dimension" of the neural network layers (i.e., the number of neurons inside each layer), but we will assume that it is bounded (for example, as a function of B and the number of predicates in the domain). We start with the definition of expresiveness of a RelNN model family.

Definition 4.4.1 (Expressiveness)**.** *We say a model family $\mathcal{M}_1$ is* at least expressive as *$\mathcal{M}_2$, written as $\mathcal{M}_1 \succcurlyeq \mathcal{M}_2$, if for all $M_2 \in \mathcal{M}_2$, there exists $M_1 \in \mathcal{M}_1$ such that M_1 can realize M_2. A model family $\mathcal{M}_1$ is* more expressive than *$\mathcal{M}_2$, written as $\mathcal{M}_1 \succ \mathcal{M}_2$, if $\mathcal{M}_1 \succcurlyeq \mathcal{M}_2$ and $\exists M_1 \in \mathcal{M}_1, \forall M_2 \in \mathcal{M}_2$, M_2 can not realize M_1.*

Arity Hierarchy We first aim to quantify how the maximum arity B of the network's representation affects its expressiveness and find that, in short, even if the inputs and outputs of neural networks are of low arity, the higher the maximum arity for intermediate layers, the more expressive the RelNN is.

Corollary 4.4.1 (Arity Hierarchy)**.** *For any maximum arity B, there exists a depth D^* such that: $\forall D \geq D^*$, RelNN[D, B + 1] is more expressive than RelNN[D, B]. This theorem applies to both fixed-precision and unbounded-precision networks. Here, by fixed-precision, we mean that the results of intermediate layers (tensors) are constant-sized (e.g., W bits per entry). Practical GNNs are all fixed-precision because real number types in modern computers have finite precision.*

Proof sketch: Our proof slightly extends the proof of Morris *et al.* [2019]. First, the set of graphs distinguishable by RelNN[D, B] is bounded by graphs distinguishable by a D-round order-B Weisfeiler-Leman test [Leman and Weisfeiler, 1968]. If models in RelNN[D, B] cannot generate different outputs for two distinct hypergraphs G_1 and G_2, but there exists $M \in$ RelNN[$D, B + 1$] that *can* generate different outputs for G_1 and G_2, then we can construct a graph classification function f that RelNN[D, $B + 1$] (with some fixed precision) can realize but RelNN[D, B] (even with unbounded precision) cannot.[‖] This gives us the tool to quantify the minimum arity for realizing certain graph reasoning functions.

[‖]Note that the arity hierarchy is applied to fixed-precision and unbounded-precision separately. For example, RelNN[D, B] with unbounded precision is incomparable with RelNN[D, $B + 1$] with fixed precision.

Corollary 4.4.2 (Logical expressiveness of relational neural networks)**.** *Let FOC_B denote a fragment of first-order logic with at most B variables, extended with counting quantifiers of the form $\exists^{\geq n}\phi$, which state that there are at least n nodes satisfying formula ϕ.*

- *(Upper Bound) Any function f in FOC_B can be realized by RelNN$[D, B]$ for some D.*
- *(Lower Bound) There exists a function f in FOC_B such that for all D, f cannot be realized by RelNN$[D, B-1]$.*

Proof: The upper bound part of the claim has been proved by Barceló *et al.* [2020] for $B = 2$. The results generalize easily to arbitrary B because the counting quantifiers can be realized by sum aggregation. The lower bound part can be proved by applying Section 5 of Cai *et al.* [1992], in which they show that FOC_B is equivalent to a $(B-1)$-dimensional WL test in distinguishing non-isomorphic graphs. Given that RelNN$[D, B-1]$ is equivalent to the $(B-2)$-dimensional WL test of graph isomorphism, there must be an FOL_B formula that distinguishes two non-isomorphic graphs that RelNN$[D, B-1]$ cannot. Hence, FOL_B cannot be realized by RelNN$[\cdot, B-1]$.

The lower bound has been proved in Chapter 2. Here, we recap the constructive proof for using RelNNs to realize FOL formulas. The breadth B is analogous to the number of variables in FOL for encoding the value of the expression; the depth D is the number of "nested quantifiers." For example, the formula $\exists x.\forall y.\forall z.p(x, y, z)$ needs 3 layers, one for each quantifier. Furthermore, we call a *RelNN* finite breadth (depth) if B (D) is independent of the number of objects. Otherwise, we call it unbounded breadth (depth).

Unfortunately, the dependence of the expressiveness of RelNNs on depth D is much more complicated. In particular, we consider RelNNs having a depth that is dependent on the number of nodes $n = |V|$ in the graph. This, in many cases, can substantially increase their expressive power [Dehghani *et al.*, 2019]. In the following, we define a *depth hierarchy* by analogy to the *time hierarchy* in computational complexity theory [Hartmanis and Stearns, 1965], and we extend our notation to let RelNN$[O(f(n)), B]$ denote the class of adaptive-depth RelNNs in which the growth-rate of depth D is bounded by $O(f(n))$.

Conjecture 4.4.1 (Depth hierarchy)**.** *For any maximum arity B, for any two functions f and g, if $g(n) = o(f(n)/\log n)$, that is, f grows logarithmically more quickly than g, then fixed-precision RelNN$[O(f(n)), B]$ is more expressive than fixed-precision RelNN$[O(g(n)), B]$.*

There is a closely related result for the *congested clique* model in distributed computing, where Korhonen and Suomela [2018] proved that $\text{CLIQUE}(g(n)) \subsetneq \text{CLIQUE}(f(n))$ if $g(n) = o(f(n))$. This result does not have the $\log n$ gap because the congested clique model allows $\log n$ bits to transmit between nodes at each iteration, while fixed-precision RelNNs allows only a constant number of bits. The reason why the result on the congested clique can

not be applied to fixed-precision RelNNs is that congested clique assumes an unbounded precision representation for each individual node.

However, Conjecture 4.4.1 is not true for RelNNs with unbounded precision, because there is an upper bound depth $O(n^{B-1})$ for a model's expressiveness power. That is, an unbounded-precision RelNN can not achieve stronger expressiveness by increasing its depth beyond $O(n^{B-1})$.

4.4.2 Compilation of BWD and S-GRS

Lemma 4.4.2 states an "equivalence" between the expressive power of relational neural networks and first-order logic formulas. In the following, we take advantage of this equivalence to compile search algorithms into relational neural networks. First, we have the following theorem.

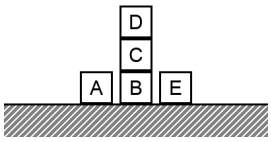
Theorem 4.4.1 (Compilation of BWD)**.** *Given a planning problem P, let T be the length of the optimal trajectory (the planning horizon), k_{BWD} be the maximum number of atoms in the goal set in* BWD*, and β be the maximum arity of atoms in the domain. The backward search algorithm* BWD *can be compiled into a relational neural network in $RelNN[O(T), \beta \cdot k_{BWD}]$.*

Proof sketch. We provide a construction in which the *RelNN* computes a set of subgoals (i.e., a set of sets of ground atoms) $Goal^d$ at each layer d. Initially, $Goal^0 = \{\{g\}\}$. Then, $sg \in Goal^d$ if there is a path of length d from any state s that satisfies $sg \subseteq s$ to g. See Appendix D.2.1 for the full proof.

Although this construction is general and powerful, it is unrealistic for large problems because the depth can be exponential in the number of objects, and the number of atoms in the subgoal conjunctive formulas can be exponential in the depth. Therefore, in the following, we will leverage the idea of serialized goal regression to make a more efficient construction.

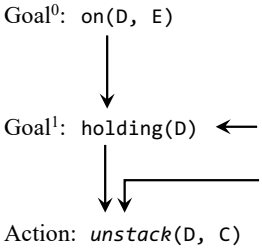
Theorem 4.4.2 (Compilation of S-GRS)**.** *Given a planning problem P of SOS width k, let T be the length of the optimal trajectory, and β be the maximum arity of atoms in the domain. The serialized goal regression search S-GRS can be compiled into a $RelNN[O(T), (k+1) \cdot \beta]$, where $k + 1 \leq k_{BWD}$. See Appendix D.2.2 for the full proof using a similar technique as in Theorem 4.4.1.*

Remark. The compilation in Theorem 4.4.2 can generate finite-breadth, unbounded-depth *RelNN* circuits for more problems than the compilation in Theorem 4.4.1. For example, in Blocks World, the number of atoms in the goal sets in BWD is unbounded. However, since the problem is of constant SOS width, it can be compiled into a finite-breadth circuit. When




```
on-table(x) ⇝  if True: apply: holding(x) || place-table(x)
on(x, y)    ⇝  if True: apply: clear(y), holding(x) || stack(x, y)
holding(x)  ⇝  if exists y. on(x, y):
                 apply: on(x, y), clear(x), handsfree || unstack(x, y)
               if on-table(x):
                 apply: on-table(x), clear(x), handsfree || pick-table(x)
clear(x)    ⇝  if holding(x): apply: holding(x) || place-table(x)
               if exists y. on(y, x):
                apply: on(y, x), clear(y), handsfree || unstack(y, x)
```

(a) The regression rules selector for Blocks World (applicable w.r.t. any constraints).



$\forall x.\ \textit{on-table-g}^{d}(x) \wedge \textit{holding}(x) \rightarrow \textit{place-table}(x)$
$\forall x.\ \textit{on-table-g}^{d}(x) \wedge \neg\textit{holding}(x) \rightarrow \textit{holding-goal}^{d+1}(x)$
$\forall x.\forall y.\ \textit{on}^{d}(x, y) \wedge \textit{clear}(y) \wedge \textit{holding}(x) \rightarrow \textit{stack}(x, y)$
$\forall x.\forall y.\ \textit{on}^{d}(x, y) \wedge \textit{clear}(y) \wedge \neg\textit{holding}(x) \rightarrow \textit{holding-goal}^{d+1}(x)$
$\forall x.\forall y.\ \textit{on}^{d}(x, y) \wedge \neg\textit{clear}(y) \rightarrow \textit{clear-goal}^{d+1}(y)$
$\forall x.\forall y.\ \textit{holding-g}^{d}(x) \wedge \textit{on}(x, y) \wedge \textit{clear}(x) \wedge \textit{handsfree} \rightarrow \textit{unstack}(x)$
$\forall x.\forall y.\ \textit{holding-g}^{d}(x) \wedge \textit{on}(x, y) \wedge \textit{clear}(x) \wedge \neg\textit{handsfree} \rightarrow \textit{handsfree-g}^{d+1}()$
$\forall x.\forall y.\ \textit{holding-g}^{d}(x) \wedge \textit{on}(x, y) \wedge \neg\textit{clear}(x) \rightarrow \textit{clear-g}^{d+1}(x)$
.....

(b) Compilation of rule selectors into First-Order Logic formulas.

Figure 4-3: **State-dependent regression rule selector in the Blocks World domain.** For brevity, we have omitted atoms in the constraint set. All rules listed above are applicable under any constraints.

the search horizon T is finite, both the SOS width and k_{BWD} will be finite because there are only a constant number of actions that can be applied to update the goal set / constraint set. So a problem can be compiled into a finite-depth, finite-breadth *RelNN* circuit with BWD compilation if and only if it can be compiled with S-GRS compilation, although S-GRS compilation may generate smaller circuits.

4.4.3 Compilation of S-GRS with a Regression Rule Selector

Unfortunately, both constructions in the previous section require a depth-$O(T)$ *RelNN* circuit, which is impractical for most real-world problems, as T can be exponential in the number of objects in a domain. In order to compile the goal regression search algorithm into a smaller circuit, we consider scenarios where the rules used to construct the (optimal) plan can be computed without doing a full search. Here the idea is to bring in human priors or machine learning to learn a low-complexity *regression rule selector* that directly predicts which goal regression rule to use. There exists such a rule selector for some problems (e.g., Blocks World) but not others (e.g., Sokoban).

Formally, we define a state-dependent *regression rule selector* (RRS) $\textit{select}(s, g, \textit{cons})$, which returns an ordered list of preconditions and a ground action a. Its function body can be written in the following form: $g^{\textit{cons}} \leftsquigarrow \text{if } \rho(s, \textit{cons}) : p_1^{\textit{cons}}, p_2^{\textit{cons}\cup\{p_1\}}, \cdots, p_m^{\textit{cons}\cup\{p_1,\cdots,p_{m-1}\}} \parallel a$, which reads: in order to achieve g under constraint *cons* at state s, if $\rho(s, \textit{cons})$ is true, we can apply the regression rule $p_1, p_2, \cdots, p_m \parallel a$. Formally, *select* is composed of a collection

of rules. Each rule is a tuple of $\langle args, \rho, pre, a, g, cons \rangle$, where *args* is a set of variables, ρ is a FOL formula that can be evaluated at any given state s, *pre* is an ordered list of preconditions, a is a ground action, g is the goal atom, and *cons* is the constraint. ρ, pre, a, g, and *cons* may contain variables in *args*. Such a rule selector can be implemented by hand (as a set of rules, as illustrated in Fig. 4-3a for the simple BlocksWorld domain), or learned by a *RelNN* model.

The main computational advantage is that now, for a given tuple $(s, g, cons)$, there is a single ordered list of preconditions and final action. Therefore, given the function *select*, we can simply construct a sequence of actions that achieves the goal by recursively applying the rule selector. In this section, we will first discuss scenarios where such rule selectors can be computed with shallow *RelNN* circuits.

Circuit complexity of regression rule selectors. We first discuss scenarios where there exist finite-depth, finite-breadth circuits for computing the regression rule selector. Note that there are three "branching" factors in choosing the regression rule: 1) the operator to use, 2) the order of the preconditions, and 3) the binding of "free" variables that are not mentioned in the goal atom.

First, in many cases, a simple condition on the state determines which operator is appropriate to use, and the preconditions can be achieved in a fixed order. Second, in many cases, the unbound variables in a rule are either determined by early preconditions or can be bound arbitrarily. When these values must be chosen very carefully (e.g.,when there are resource constraints), then a simple regression rule selector may not exist. For example, in Blocks World, to achieve *clear*(B), then the optimal way is to perform $unstack(A, B)$, where A is the object on B. Furthermore, in this case, the precondition order is fixed: $clear(\mathrm{B}) \leftsquigarrow on(\mathrm{A}, \mathrm{B}), clear(\mathrm{A}), handsfree() \parallel unstack(\mathrm{A}, \mathrm{B})$ for arbitrary pairs of objects A, B. In the following, we consider cases where the regression rule selector can be computed with a first-order logic formula, therefore a finite-depth and finite-breadth *RelNN* circuit.

Serializability of RRS. Given a regression rule selector *select*, we can obtain a single trajectory (if g is reachable from s under constraints *cons*). We define $tr^{select}(s, g, cons)$ as the trajectory returned by recursive application of *select*. It returns $\perp$ if no plan can be found.

Definition 4.4.2 (RRS Serializability)**.** *A regression rule selector is serializable if and only if the following condition holds for any state s, any goal g, and any constraint set. Consider the regression rule returned by* $select(s, g, cons) \leftsquigarrow p_1, p_2, \cdots, p_k \parallel a$. *If g is achievable from s, then it can be achieved via the concatenation of the following trajectories:* $\overline{a}_1 = tr^{regress}(s, p_1, constraint)$, $\overline{a}_2 = tr^{regress}(\mathcal{T}(s, \overline{a}_1), p_2, constraint \cup \{p_1\})$, $\cdots$, $\overline{a}_k$, *and* $\{a\}$.

Compiling policy neural networks. If there exists a regression rule selector that is computable with a finite-depth, finite-breadth *RelNN* circuit, it will be possible to construct a policy for the original problem with another *RelNN* circuit. This construction can be *dramatically* more efficient than the general constructions in Section 4.4.2. Let k denote the "width" of the regression rule selector, i.e., the maximum size of the constraints we need to keep track of when applying *select*.

Theorem 4.4.3 (Compilation of S-GRS with a regression rule selector)**.** *Given a regression rule selector* select *(i.e., the state-constraint condition function ρ) that can be computed by a relational neural network in RelNN$[D_r, B_r]$, for any planning problem P, if $tr^{select}(s_0, g, \emptyset) \neq \bot$, letting d be the depth of the regression tree, then there is a RelNN circuit for the problem in RelNN$[O(d \cdot D_r), \max(B_r, \beta)]$, where β is the maximum arity of predicates.*

Proof. For any step i, let $pre_{\leq i}$ be the length i prefix of the precondition list. We define $Goal^d$ to be the single goal atom and set of constraints we need to satisfy at depth d. We have the following rules:

$$\forall d. \forall i < |pre|. \Big(Goal^d = (g, cons)\Big) \wedge \rho(s, cons) \wedge pre_{\leq i} \wedge \neg pre_{i+1} \Rightarrow \Big(Goal^{d+1} = (pre_{i+1}, cons \cup pre_{\leq i})\Big)$$
$$\forall d. \Big(Goal^d = (g, cons)\Big) \wedge \rho(s, cons) \wedge pre \Rightarrow a$$

Here the evaluations of preconditions *pre* are based on the current state (input to the policy network) and they can be evaluated in parallel. We illustrate this construction in Fig. 4-3b. The first set of rules computes the next subgoal (and constraint set) to achieve, while the second set of rules outputs the next action when all preconditions of the current rule have been satisfied (which will be the output of the entire policy). Since at each layer d, we only need to keep track of one tuple of $(g, cons)$, the breadth of the circuit is $\max(B_r, \beta)$, where β is required to store the input state. □

This construction yields a significant reduction in breadth (from tracking all conjunctions up to k_{BWD} atoms to only one tuple of $(g, cons)$), as well as a reduction of the number of layers for the *RelNN* policy, depending on the domain. In particular, if the average number of preconditions that need to be recursively solved in each rule is b (roughly equal to the number of preconditions in each rule), the depth of the regression tree is $d = O(\log_b T)$, where T is the planning horizon.

Finally, for some problems, even if we have a regression rule selector, to compute a plan, we may still need an unbounded depth of search steps (i.e., d depends on the number of objects in the domain). For example, in Blocks World, to achieve *clear*(A), the number of steps needed is proportional to the number of objects on top of A. There are two ways to

Problem	Constant Breadth?	Depth
BlocksWorld	✓ ($k = 1$)	Unbounded
Logistics	✓ ($k = 0$)	Unbounded
Gripper	✓ ($k = 0$)	Constant
Rover	✓ ($k = 0$)	Unbounded
Elevator	✓ ($k = 0$)	Unbounded
Sokoban	✗	Unbounded

Table 4.1: **Width and circuit complexity analysis for 6 problems widely used in AI planning communities.** For problems with a constant circuit breadth, we also annotate their regression width k. "Unbounded" depth means that the depth depends on the number of objects.

Task	Model	n=10	n=30	n=50
As3	*RelNN*$[1, 2]$	$0.2_{\pm 0.05}$	$0.02_{\pm 0.01}$	$0.0_{\pm 0.0}$
As3	*RelNN*$[2, 2]$	$\mathbf{1.0}_{\pm 0.0}$	$\mathbf{1.0}_{\pm 0.0}$	$\mathbf{1.0}_{\pm 0.0}$
Log.	*RelNN*$[3, 2]$	$1.0_{\pm 0.0}$	$0.30_{\pm 0.03}$	$0.23_{\pm 0.05}$
Log.	*RelNN*$[f(n), 2]$	$\mathbf{1.0}_{\pm 0.0}$	$\mathbf{1.0}_{\pm 0.0}$	$\mathbf{1.0}_{\pm 0.0}$
BW	*RelNN*$[3, 2]$	1.0 / $2.9_{\pm 0.4}$	1.0 / $10.2_{\pm 0.45}$	1.0 / $15.7_{\pm 2.5}$
BW	*RelNN*$[f(n), 2]$	$\mathbf{1.0}$ / $\mathbf{2.5}_{\pm 0.4}$	$\mathbf{1.0}$ / $\mathbf{3.1}_{\pm 0.5}$	$\mathbf{1.0}$ / $\mathbf{3.5}_{\pm 0.4}$

Table 4.2: **Success rate of learned policies in different environments.** For Assembly3 (As3) and Logistics (Log.), we show the success rate. For Blocks World (BW), we show the success rate / average solution length. We choose $f(n) \stackrel{\text{def}}{=} n/5 + 1$ for Logistics and $f(n) \stackrel{\text{def}}{=} n/10 + 3$ for BlocksWorld. In the notation of *RelNN*$[D, B]$, D is the number of layers and B is the maximum arity of edges.

build a relational neural network policy in this case. First, we can allow the network to have a variable number of iterations by applying the same layer recurrently several times. Second, there might be "shortcuts" that we can learn. We discuss both solutions in Appendix D.2.3. These methods allow us to construct finite-depth *RelNN* circuits for a variety of domains, including Blocks World, and path-finding in any acyclic maze.

4.5 Problem Analysis and Results

We first analyze the regression width and circuit complexities for familiar AI planning problems. Table 4.1 summarizes the results on a simplified setting where the goal of the planning problem is a single atom. In summary, most of these problems have a constant breadth (i.e., the regression width of the corresponding problem is constant) except for Sokoban. Most problems have an unbounded depth: that is, the depth of the circuit will

depend on the number of objects (e.g., the size of the graph or the number of blocks). For problems in this list, when there are multiple goals, usually the goals are not serializable (in the optimal planning case). If we only care about satisficing plans, for Logistics, Gripper, Rover, and Elevator, there exists a simple serialization for any conjunction of goal atoms (basically achieving one goal at a time). See Appendix D.3 for more detailed proofs and additional discussions of generalization to multiple goals and suboptimal plans.

Next, given this analytical understanding of the relationship between planning problems and their policy circuit complexity, we perform some simple experiments to see whether that relationship is borne out in practice. Relational neural networks have been demonstrated to be effective in solving some planning problems [Chapter 2; Jiang and Luo, 2019; Li *et al.*, 2020c], but their complexity has not been systematically explored. We consider two families of problems: one predicted to require finite depth and one predicted to require unbounded depth. For all tasks, we use neural logic machines introduced in Chapter 2 as the model; we set the number of hidden neurons in the MLP layers to be sufficiently large (64 in all experiments) so that it is not the bottleneck for network expressiveness. We use the same encoding style for all problems (the graph-like relationship). Therefore, whether different problems have bounded or unbounded width primarily depends on the available goal regression rules, how these rules can be serialized, and the width of the rules. We show the average performance across 3 random seeds, with standard errors.

Assembly3: finite depth circuits. The domain Assembly3 contains n objects. Each object has a category, chosen from the set $\{A, B, C\}$. The goal of the task is to select three objects o_A, o_B, and o_C sequentially, one for each category, while satisfying a matching constraint: $match(o_A, o_B)$ and $match(o_B, o_C)$. Therefore, to select the first object (e.g., an object of type A), the policy needs to perform two layers of goal regression (first find the set of possible B-typed objects that match the object, and then find another C-typed object). We trained two models. Both models have breadth 2, but the first model has only 1 layer (theoretically not capable of representing the two-step goal regression), while the second model has 2 layers. Shown in Table 4.2, we train models on environments with 10 objects and test them on environments with 10, 30, and 50 objects. The first model is not able to learn a policy for the given task, while the second exhibits perfect generalization.

Logistics: unbounded depth circuits. We also construct a simple Logistics domain with only cities and trucks (no airplanes), in which the graphs are not strongly connected (by first sampling a directed tree and then adding forward-connecting edges). We train the policy network on problems with less than n=10 cities and test it on n=30 and n=50. Here, we trained two policies. The first policy has a constant depth of 3, and the second policy has a

depth of $D = n/5 + 1$, which is a manually chosen function such that it is larger than the diameter of the graph. Shown in Table 4.2, the first model fails to generalize to larger graphs due to its limited circuit depth. Intuitively, to find a path in a graph by recursively applying the regression rule, we need a circuit of an adaptive depth (proportional to the length of the path). By contrast, the second model with adaptive depths generalizes perfectly.

BlocksWorld-Clear: unbounded depth circuits. Based on the goal regression analysis, in BlocksWorld, in order to achieve the goal atom *clear*(A) for a specific block A, we would need a circuit that is as deep as the number of objects on A in the initial state. We follow the same training and evaluation setup as in Logistics. Our analysis focuses on the length of the generated plan. Shown in Table 4.2, although both policy networks accomplish the task with a 1.0 success rate (because there is a trivial policy that randomly drops clear blocks onto the table), the policy whose depth depends on the number of blocks successfully finds a plan that is significantly shorter than the fixed-depth policy. This suggests the importance of using RelNNs with a non-constant depth for certain problems.

4.6 Related Work

Planning complexity. Most of the existing work on planning complexity considers the NP-Completeness or PSPACE-Completeness of particular problems, such as the traveling salesman problem [Karp, 1972], Sokoban [Culberson, 1997], and Blocks World [Gupta and Nau, 1992]. In general, the decision problem of plan existence is PSPACE-Complete for atomic STRIPS planning problems [Bylander, 1994]. Seminal work on fine-grained planning problem complexity introduces planning width [Chen and Giménez, 2007] and the IW algorithm [Lipovetzky and Geffner, 2012; Drexler *et al.*, 2022]. Our work is greatly inspired by these ideas and strives to connect them with circuit complexity.

Subgoal serialization. The concept of serialized goal regression is not completely new. These rules are particularly related to the methods in hierarchical task networks [Erol *et al.*, 1994a] and can also be seen as special cases of hierarchical goal networks [Alford *et al.*, 2016], derived by dropping preconditions [Sacerdoti, 1974]. In addition, others, including Korf [1987] and Barrett and Weld [1993], have characterized different degrees of serializability of subgoals, but their analysis is not as fine-grained as this.

Relational neural networks. Our circuit complexity analysis builds on existing work on relational neural network expressiveness, in particular, GNNs and their variants. Xu *et al.* [2019b] provides an illuminating characterization of GNN expressiveness in terms of

the WL graph isomorphism test. Azizian and Lelarge [2021] analyze the expressiveness of higher-order Folklore GNNs by connecting them with high-dimensional WL-tests. Barceló *et al.* [2020] and Luo *et al.* [2022b] reviewed GNNs from the logical perspective and rigorously refined their logical expressiveness with respect to fragments of first-order logic. Our work extends their results, asking the question of what planning problems a *RelNN* can solve.

Our work is also related to how GNNs may generalize to larger graphs (in our case, planning problems with an arbitrary number of objects). Xu *et al.* [2020, 2021a] have studied the notion of *algorithmic alignment* to quantify such structural generalization. In Chapter 6, we have provided empirical results showing that NLMs generalize in Blocks World and many other domains. Buffelli *et al.* [2022] introduced a regularization technique to improve GNNs' generalization and demonstrated its effectiveness empirically. Xu *et al.* [2021a] also showed empirically on certain algorithmic reasoning problems (e.g., max-Degree, shortest path, and the n-body problem). In this chapter, we focus on constructing policies that generalize instead of algorithms, using a similar idea of compiling specific (search) algorithms into neural networks.

4.7 Discussion

To summarize, we have illustrated a connection between classical planning width, regression width, search complexity, and policy circuit complexity. We derive upper bounds for search and circuit complexities as a function of the regression width of problems and planning horizon. These results provide an explanation of the success of relational neural networks learning generalizable policies for many object-centric domains. The compilation algorithms highlight that when there are resource constraints such as free space, agents' inventory size, etc., a small policy circuit may not exist. This includes cases such as Sokoban and general task and motion planning. Furthermore, our idea of serialization can be generalized to hierarchical decomposition of problems; it can be potentially extended to other domains such as optimization and planning under uncertainty.

Although all of the analyses in this chapter have been done for fully discrete domains by analyzing first-order logic formulas, understanding the implications in relational neural network learning is important because, ultimately, *RelNNs* can be integrated with perception and continuous action policy networks to solve realistic robotic manipulation problems (e.g., physical Blocks World [Li *et al.*, 2020c]). It is challenging to directly generalize our proof to scenarios where states and actions are continuous values because in these cases, there will be an infinite number of possible actions (e.g., the choice of grasping poses, trajectories, etc.). We believe that it is an interesting question to investigate similar "width"-like definitions in

sampling and discretization-based continuous planning algorithms. So far, we have only analyzed the circuit complexity of policies for individual planning problems; an interesting future direction is to extend these analyses to the learning and inference of "general" policies such as large language models that can generate plans for many different planning domains. This idea motivates the development of hybrid imperative–declarative representations of actions in Chapter 12.

This chapter marks the end of our exploration of neuro-symbolic models for logical reasoning. In the next chapter, we will turn to neuro-symbolic methods for constraint satisfaction solving, another important reasoning problem that typically arises in robot planning.

Chapter 5

Neuro-Symbolic Continuous Constraint Satisfaction

This chapter focuses on a different important task in neuro-symbolic representation and reasoning: learning to solve continuous constraint satisfaction problems (CCSP). Constraint satisfaction provides a powerful abstraction for many problems in robotics, as diverse physical and symbolic requirements, such as collision avoidance, kinematic feasibility, and logical preconditions, can be uniformly expressed as constraints. It is inherently compositional: solving a union of subproblems is equivalent to solving a larger constraint satisfaction problem formed by merging all of the subsets of constraints. Moreover, constraint satisfaction can also be understood as the inverse of logical deduction. Whereas deduction derives consequences from a set of rules, constraint solving seeks assignments of variables that are consistent with those rules.

Previous methods towards continuous constraint satisfaction solving primarily rely on hand-engineering or learning generators for specific constraint types and then rejecting the value assignments when other constraints are violated. By contrast, our proposed model in this chapter, the compositional *diffusion continuous constraint solver* (*Diffusion-CCSP*) derives global solutions to CCSPs by representing them as factor graphs and combining the energies of diffusion models trained to sample for individual constraint types. Diffusion-CCSP exhibits strong generalization to novel combinations of known constraints, and it can be integrated into a task and motion planner to devise long-horizon plans that include actions with both discrete and continuous parameters, which is the key problem class that will arise in robotic decision making in Part III.

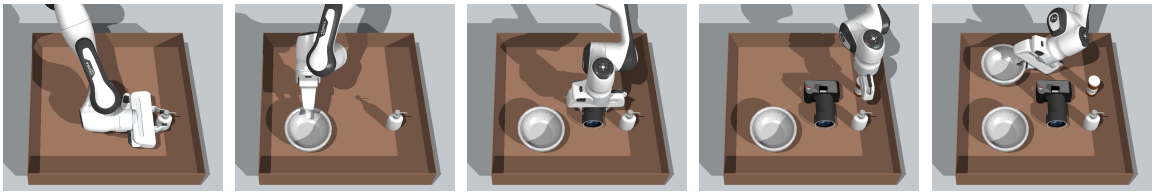
The material of this chapter has been previously published in Yang *et al.* [2023b]. Zhutian Yang and Yilun Du, in particular, contributed significantly to the materials presented in this chapter.

5.1 Introduction

Robotic manipulation planning relies critically on the ability to select continuous values, such as grasps and object placements, that satisfy complex geometric and physical constraints, such as stability and lack of collision. Existing approaches have used *separate* samplers, obtained through learning or optimization, for each constraint type. For instance, GraspNet [Mousavian *et al.*, 2019] focuses on generating valid grasps, and StructFormer [Liu *et al.*, 2022b] specializes in generating semantically meaningful object placements. However, complex problems require a general-purpose solver to produce values that conform to multiple constraints simultaneously. Consider the task of 3D object packing with a robot arm, as depicted in Figure 5-1. Given the geometric and physical properties of the objects, our objective is to generate the target pose and motion trajectories that fulfill three essential criteria. First, all objects must be contained within the container and satisfy qualitative user requirements (e.g., bowls should be placed next to each other). Second, collisions among objects and the robot, during their movement and when stationary, must be avoided. Finally, the final configuration should be physically stable.

Constructing or training a monolithic model capable of solving *every possible combination* of goals and constraints can be challenging due to limited data. Therefore, it is essential for general-purpose robot planners to *reuse* and to *compose* individual solvers for overall tasks. A common strategy is to combine specialized methods for solving individual problems in a *sequential* manner via *rejection sampling*, which is widely used by state-of-the-art planners for object rearrangement and general task and motion planning (TAMP) problems [Kaelbling and Lozano-Pérez, 2011; Garrett *et al.*, 2020a; Mao *et al.*, 2022; Paxton *et al.*, 2022; Yang *et al.*, 2023a]. The set of solvers is applied in a predetermined order guided by heuristics (e.g., first sampling object poses and then grasps and trajectories). If a previously sampled value violates a later constraint (e.g., the sampled pose in the box does not leave enough space for the gripper during object placement), backtracking is performed. This rejection-sampling-based approach can be highly inefficient when faced with many constraints. An alternative approach is to formulate the entire constraint satisfaction problem using a differentiable objective function and solve it using local optimization methods [Toussaint, 2015; Pang *et al.*, 2023]. However, these methods usually require manually-specified differentiable constraint formulas, but many important constraints, such as those corresponding to human directives like 'next to', need to be learned from data.

We propose to use constraint graphs as a unified framework for specifying constraint-satisfaction problems as novel combinations of learned constraint types and apply diffusion-model-based constraint solvers to find solutions that jointly satisfy the constraints. A constraint graph consists of nodes representing decision variables (e.g., grasping poses,



(a) The dispenser is constrained to be grasped only from the sides.

(b) The gripper pose when grasping the bowl should be reachable.

(c) The gripper at grasp poses shouldn't collide with other objects.

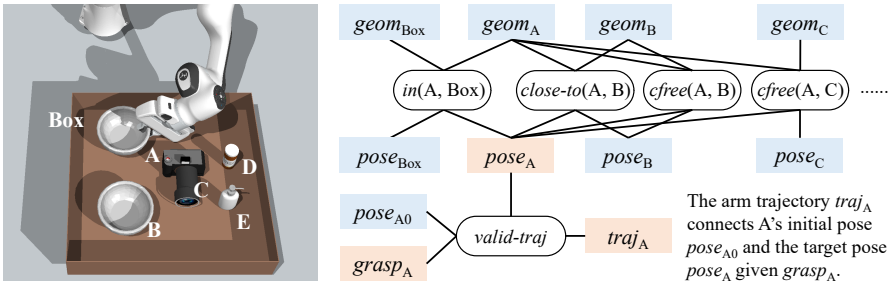
(d) All arm trajectories shouldn't collide with the object or the tray.

(e) The two bowls are constrained to be next to each other.

Figure 5-1: **Solving Continuous Constraint Satisfaction Problems by Composing Diffusion Models.** Our approach combines diffusion models, representing individual constraints, to generate object placement poses. The choice of an object's placement depends on both qualitative constraints about object placement and collision avoidance constraints on the object and the robot gripper.

placement poses, and robot trajectories) and nodes representing constraints among these decision variables. Our method, the compositional diffusion constraint solver (Diffusion-CCSP), learns a collection of diffusion models for individual types of constraints and recombines them to address novel problem instances, finding satisfying assignments via a diffusion process that generates diverse samples from the feasible region. In particular, each diffusion model learns to generate a distribution of feasible solutions for one particular type of constraint (e.g., collision-free placements). Since the diffusion models are generative models of the set of solutions, at inference time we can ***condition*** on an arbitrary subset of the variables and solve for the rest. Furthermore, the diffusion process is ***compositional***: since each diffusion model is trained to minimize an implicit energy function, the task of global constraint satisfaction can be cast as minimizing the global energy of solutions (in this case, simply the summation of individual energy functions). These two features introduce notable flexibility in both training and inference. Component diffusion models can be trained independently or simultaneously based on paired compositional problems and solutions. At test time, Diffusion-CCSP generalizes to novel combinations of known constraint types and graphs with more variables than those seen during training.

We evaluate Diffusion-CCSP on four challenging domains: 2D triangle dense-packing, 2D shape arrangement with qualitative constraints, 3D shape stacking with stability constraints, and 3D object packing with robots. Compared with baselines, our method exhibits efficient inference time and strong generalization to novel combinations of constraints and problems with more constraints.



(a) Visualization of the environment while placing object A.

(b) Visualization of the constraint graphs associated with the object placement. There are three decision variables.

Figure 5-2: **Continuous constraint satisfaction problem (CCSP) in robot planning.** It unifies geometric, physical, and qualitative constraints. To place A into the tray, we need to generate the grasping pose $grasp_A$, placement pose $pose_A$, and the robot arm trajectory. We omit collision-free constraints with robots in (b) for brevity.

5.2 Compositional Diffusion Constraint Solvers

Our goal is to develop a general algorithm capable of learning and solving diverse *continuous constraint satisfaction problems* (CCSPs) encountered in robotic manipulation tasks. To achieve this, we adopt a generic graph-based representation of CCSPs unifying various geometric, physical, and qualitative constraints*.

5.2.1 Formulation and Representation of Constraint Satisfaction Problems

Formally, a *continuous constraint satisfaction problem* (CCSP) can be represented as a graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{U}, C \rangle$. Each $v \in \mathcal{V}$ is a decision variable (such as the pose of an object), while each $u \in \mathcal{U}$ is a conditioning variable (such as the geometry of an object, which will be constant at performance time). Each $c \in C$ is a constraint, formally a tuple of $\langle t_c, n_c \rangle$, where t_c is the type of the constraint (e.g., collision-free), and $n_c = (\mathcal{V}^c, \mathcal{U}^c)$ contains two sets of variables in $\mathcal{V}$ and $\mathcal{U}$, which correspond to the arguments to this constraint. For example, a collision-free constraint between object A and B can be represented as $\langle \text{cfree}, (pose_A, pose_B, geom_A, geom_B) \rangle$. Here, $pose_A$ and $pose_B$ are decision variables that represent the target pose of objects A and B, while $geom_A$ and $geom_B$ are conditioning variables that represent the geometry of objects A and B.

Let V and U be assignments of values to the decision and conditioning variables $\mathcal{V}$ and $\mathcal{U}$, respectively, and let V^c and U^c be the subsets of assigned values to variables in

*Throughout the chapter, we use the word qualitative constraint to refer to constraints that can not be easily and accurately described analytically—in particular, spatial relationships such as "left" and "close-to." These constraints depend on contexts and would ideally be learned from human annotations.

constraint c. For example, in the 3D packing task shown in Figure 5-2a, the assignment V to variables $\mathcal{V}$ is a mapping from the variables $pose_i$ to an $SE(3)$ pose of object i. The solution should satisfy that for all $c = \langle t_c, (\mathcal{V}^c, \mathcal{U}^c) \rangle \in C$, $test(t_c, (V^c, U^c)) = 1$, where *test* takes the constraint type and the assignments to each variable and returns 1 if the constraint c is satisfied and 0 otherwise. The function *test* can be implemented as human-specified rules, learned classifiers, or the outcome of a physics simulator.

Example. Figure 5-2 illustrates the constraint graph associated with the pick-and-place of a bowl (A) into a box close to another bowl (B), while avoiding it and other obstacles. It includes three decision variables: $grasp_{\mathrm{A}}$ (transform between object and hand), $pose_{\mathrm{A}}$ (target pose), and $traj_{\mathrm{A}}$ (robot trajectory). The scenario involves three constraint groups: 1) qualitative constraints, including $in(\mathrm{A}, \mathrm{Box})$ ensuring the bowl's placement within the box and $close\text{-}to(\mathrm{A}, \mathrm{B})$ asserting proximity between two bowls; 2) collision avoidance constraints such as $cfree(\mathrm{A}, \mathrm{B})$, which asserts that objects A and B do not collide in the final placement; 3) trajectory constraints, encapsulated by *valid-traj*, which assert that the trajectory $traj_{\mathrm{A}}$ is a feasible robot path that connects the bowl's initial position $pose_{\mathrm{A0}}$ and the target $pose_{\mathrm{A}}$, given $grasp_{\mathrm{A}}$. It is important to note that these constraints are correlated: for example, the choice of $pose_A$ requires the consideration of the robot trajectory to ensure the existence of a valid grasping pose and trajectory.

5.2.2 Compositional Diffusion Models

Given a set of constraints C, decision variables $\mathcal{V}$, and conditioning values $\mathcal{U} = U$, we wish to find an assignment of V_0 that satisfies C. We represent the conditional distribution of variable assignments given an individual constraint c using a diffusion model $p_c(\mathcal{V}^c = V^c \mid \mathcal{U}^c = U^c) \propto \mathbb{1}[test(t_c, (V^c, U^c))]$, which is effectively uniform over the (bounded) region of $\mathcal{V}^c$ that satisfies the constraint. Finding a satisfactory assignment of variables V_0 then corresponds to finding an assignment that maximizes the likelihood in the joint distribution constructed from the product of the individual constraint models. Each constraint diffusion model represents $p(\mathcal{V}^c \mid \mathcal{U}^c)$ in the form of an energy-based model, $p(\mathcal{V}^c \mid \mathcal{U}^c) \propto e^{-E(\mathcal{V}^c \mid \mathcal{U}^c)}$ through its score [Liu *et al.*, 2022a; Du *et al.*, 2023], so the probability maximization problem corresponds to the energy minimization problem:

$$V_0 = \arg\max_V \prod_{c \in C} p_c(\mathcal{V}^c \mid \mathcal{U}^c) = \arg\min_V \sum_{c \in C} E(\mathcal{V}^c \mid \mathcal{U}^c) \ .$$

We solve this energy minimization problem by using a sampler based on the annealed unadjusted Langevin algorithm (ULA) [Roberts and Tweedie, 1996; Du *et al.*, 2023]. The ULA algorithm is a special case of Langevin Monte Carlo — it iteratively optimizes a

Algorithm 4 Training

repeat
  $\mathcal{G}, V_0 \sim \mathcal{D}$
  $t \sim \text{Uniform}(\{1, \ldots, T\})$; $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
  Take gradient descent step on
    $\nabla_\theta \left\| \epsilon - \sum_{c \in C} \epsilon_{t_c}(\sqrt{\bar{\alpha}_t} V_0^c + \sqrt{1 - \bar{\alpha}_t}\epsilon, t)) \right\|^2$
until converged

Algorithm 5 Sampling

$V_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
for $t = T, \ldots, 1$ **do**
  if $t > 1$ **then** $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ **else** $\mathbf{z} = \mathbf{0}$
  $V_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(V_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \sum_{c \in C} \epsilon_{t_c}(V_t^c, U^c, t)\right) + \sigma_t \mathbf{z}$
return V_0

noisy sample using the gradient of the energy function, with added noise at each step of optimization.

During training, we optimize a noise prediction model ϵ_t for each constraint type t. Each ϵ_t takes conditioning variables U^c and a noisy version of V^c as input, and predicts the noise applied on V^c. Most simply, we could train the diffusion models independently for each constraint, and combine them at prediction time. In our experiments, we train multiple diffusion models simultaneously: since each training example corresponds to some constraint graph, it is natural to train all the constraints that are instantiated in that graph jointly from the example. Note that the constraint graphs within the training set and between training and testing need not be the same—they just need to be constructed from the same basic set of constraint types.

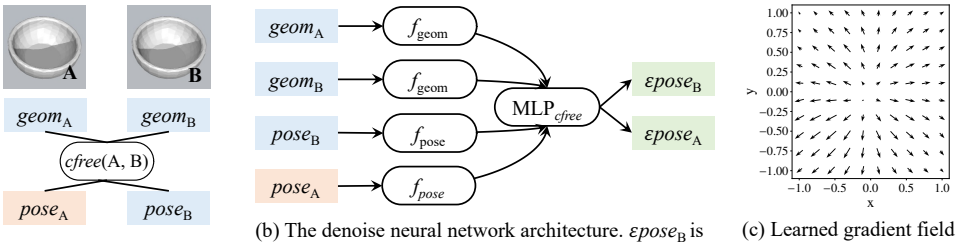
Specifically, our training dataset $\mathcal{D}$ contains a set of constraint graphs, each with conditioning values and a solution to the decision variables: $\langle \mathcal{G} = (\mathcal{V}, \mathcal{U}, C), U, V_0 \rangle$. For each type of constraint, we randomly initialize a denoising function ϵ_t, and sum the denoising predictions over the constraints in the graph to denoise a noisy version of V_0:

$$\mathcal{L}_{\text{MSE}} = \mathop{\mathbf{E}}_{\substack{\langle \mathcal{G}=(\mathcal{V},\mathcal{U},C),U,V_0 \rangle \in \mathcal{D} \\ \epsilon \sim \mathcal{N}(\mathbf{0},\mathbf{I}), t \sim \text{Uniform}(1,\cdots,T)}} \left[\left\| \epsilon - \sum_{c \in C} \epsilon_{t_c} \left(\sqrt{\bar{\alpha}_t} V_0^c + \sqrt{1 - \bar{\alpha}_t}\epsilon, U^c, t \right) \right\|^2 \right],$$

where ϵ is random Gaussian noise applied to V_0, α_t are predefined noise scheduling for training and sampling following Nichol and Dhariwal [2021], $\bar{\alpha}_t := \prod_{i=1}^{t} \alpha_t$, t is a randomly sampled diffusion step, and T is the total number of diffusion steps. The full pseudocode for training and sampling from our conditional diffusion models for constraints can be found in Algorithms 4 and 5.

5.2.3 Geometric and Physical Constraint Solving with Diffusion-CCSP

Given the general formulation of Diffusion-CCSP, we describe its concrete implementation in our robot planning setting. Recall that for each type of constraint, we have defined a



(a) The *cfree* constraint.

(b) The denoise neural network architecture. $\epsilon pose_B$ is ignored because $pose_B$ is fixed in this case.

(c) Learned gradient field for the *cfree* constraint.

Figure 5-3: **Illustration of the denoising neural network in Diffusion-CCSP.** We use the constraint *cfree* (collision-free) as the example and show the gradient map learned by Diffusion-CCSP for the centroid of B at (x, y) when A is at $(0, 0)$.

"denoise" function ϵ_t that takes the conditioning values and (the noisy version of) the decision values as inputs, and predicts the noise applied on the decision values. In our robotic domains, we focus on two variable types: geometry variables and pose/transformation variables. We illustrate the neural network architecture for collision-free constraints though the construction easily generalizes to other constraint types.

Recall that the argument to a collision-free constraint c has four elements: $\mathcal{U}^c = (geom_A, geom_B)$, $\mathcal{V}^c = (pose_A, pose_B)$. We use $pose^t_A$ to denote the value of this variable at diffusion time step t. The output of ϵ_{cfree} is a tuple $(\epsilon pose^t_A, \epsilon pose^t_B)$ which corresponds to the noise applied on $pose_A$ and $pose_B$ at diffusion time step t. In our implementation, an object's shape is represented as a 3D axis-aligned bounding box, and its pose is represented as a 5-dimensional feature vector composed of the 3-DoF translation and rotation γ about $+z$ axis, encoded using $\sin(\gamma)$ and $\cos(\gamma)$. We use a multi-layer perceptron (MLP) f_{geom} to encode the object shapes (shared across all constraints) and a constraint-specific MLP f_{pose} to encode poses. Both geometric features and pose encoders map input representations into a fixed-length vector embedding. The function ϵ_{cfree} is implemented as

$$\epsilon^t_{\text{cfree}}(geom_A, geom_B, pose_A, pose_B) = g_{\text{cfree}}\left(f_{\text{geom}}(geom_A) \oplus f_{\text{geom}}(geom_B) \oplus f_{\text{pose}}(pose_A) \oplus f_{\text{pose}}(pose_B) \oplus t\right),$$

where $\oplus$ denotes vector concatenation, and g_{cfree} is another MLP model that takes the concatenated input-feature vectors and outputs a vector of length 10. The first 5 entries correspond to the prediction of $\epsilon pose^t_A$, and the rest correspond to $\epsilon pose^t_B$, where t is the decoding time step.

Example. Figure 5-3 illustrates the neural network architecture for one specific constraint and the corresponding gradient map learned by Diffusion-CCSP. In this example, our objective is to generate a pose for object A that avoids collisions with object B in its current position. The neural network encodes the four input features and predicts the noise for $pose_A$

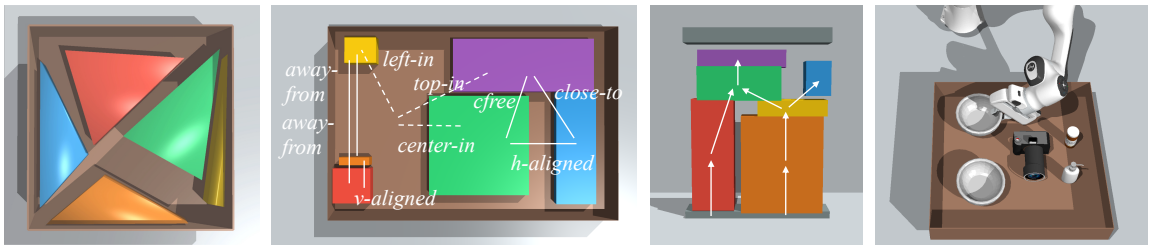
and $pose_{\text{B}}$. As the pose of the object B remains unchanged, the noise (gradient) predicted by ϵ_{cfree} for its pose is disregarded. Figure 5-3c visualizes the gradient map for the X and Y components of $pose_{\text{A}}$, where the gradient points approximately outward from the current location of object B. It is important to note that the gradient over the pose variable $pose_{\text{A}}$ will have contributions from multiple constraints in the constraint graph. See the gradient fields of all qualitative constraints in Appendix E.6.

Planning. So far, we have presented a generic solution to solving constraint satisfaction problems that involve geometric and physical constraints, assuming that the constraint graph is given as input to the algorithm. This approach can be directly used to solve particular tasks such as the pose prediction task in object rearrangements. Our primary motivation is to solve subproblems that arise during general robot manipulation planning. In Appendix E.5, we illustrate how Diffusion-CCSP can be integrated with a search algorithm to solve general task and motion planning (TAMP) problems, where the constraint graphs are automatically constructed based on the sequence of actions that has been applied and the goal specification of the task.

5.3 Experiments

We evaluate Diffusion-CCSP through CCSP tasks incorporating geometric, physical, and qualitative constraints. We include all environmental details and sample complexity discussions in Appendix E.1. For implementation and training details and additional discussions on model training and inference, see Appendix E.2. For analysis of Diffusion-CCSP's failure cases in each task, see Appendix E.3.

We compare our full model to one model variant, a sequential-sampling baseline, and StructDiffusion [Liu *et al.*, 2023b]. ***Diffusion-CCSP w. Reverse Sampling***: In this variant, we directly use the standard reverse diffusion process to sample from composed diffusion models [Liu *et al.*, 2022a]. This method is faster at sampling than ULA, but is generally found to have worse generalization performance. We include a detailed comparison between these two in Appendix E.4. ***Sequential Sampling***: We sequentially sample each decision variable according to a generic sampler (e.g., for 2D poses, we randomly sample a location that is within the tray and a random rotation) and check all geometric constraints (e.g., *in* and *cfree* associated with the decision variable. For each variable, we sample 50 samples, and report failure when no successful samples can be produced within 50 samples. ***Struct-Diffusion***: StructDiffusion is a transformer-based diffusion model for predicting object placement poses based on object shapes and a single scene-level constraint. It takes a set of object shape representations, the noisy sample of their poses, and the diffusion time step,



(a) 2D Triangle Packing. (b) 2D Shape Arrangement with Qualitative Constraints (c) 3D Object Stacking with Stability Constraints. (d) 3D Object Packing with Robots.

Figure 5-4: **Illustration of four domains.** (a) Triangle packing. (b) Dense 2D packing with qualitative constraints. The figure shows a subset of 45 constraints of 13 types. (c) 3D object stacking. The arrows show the support relationships. (d) 3D object packing with a panda robot.

and predicts the noise on object poses. Since it can only handle a single global constraint, it is only applicable in two of our tasks (Section 5.3.1 and Section 5.3.4).

5.3.1 2D Triangle Packing

The first domain considers packing a set of random triangles into a square tray, ensuring no overlap. Triangles can be rotated, and their shapes are encoded by their vertex coordinates in their zero poses. The model's output includes a 2D translation vector and a 1D rotation. Training data consists of 30,000 solutions to problems with 2-4 triangles, and testing includes 100 problems with 2-5 triangles. We generate training solutions by randomly splitting the tray.

Fig. 5-5a shows the result, and Fig. 5-4a shows a concrete testing problem and its solution found by Diffusion-CCSP. When the number of triangles becomes large, it becomes increasingly difficult for the rejection-sampling baseline to succeed. The transformer architecture used in StructDiffusion successfully models the relational structure among all pairs of objects. Therefore, it performs similarly to our method and has a better generalization to out-of-distribution CCSPs in this task.

5.3.2 2D Shape Arrangement with Qualitative Constraints

This domain involves packing rectangles into a 2D box, satisfying geometric and qualitative constraints resembling scenarios such as dining table settings and office desk arrangements. Qualitative constraints include *center-in*(A, Box), *left-in*(A, Box), *left-of*(A, B), *close-to*(A, B), *vertically-aligned*(A, B), etc., along with the geometric ones such as containment (*in*) and collision-free (*cfree*). Different problem instances have different box sizes and constraints. The training dataset consists of 30,000 problems and solutions with 2 to 4

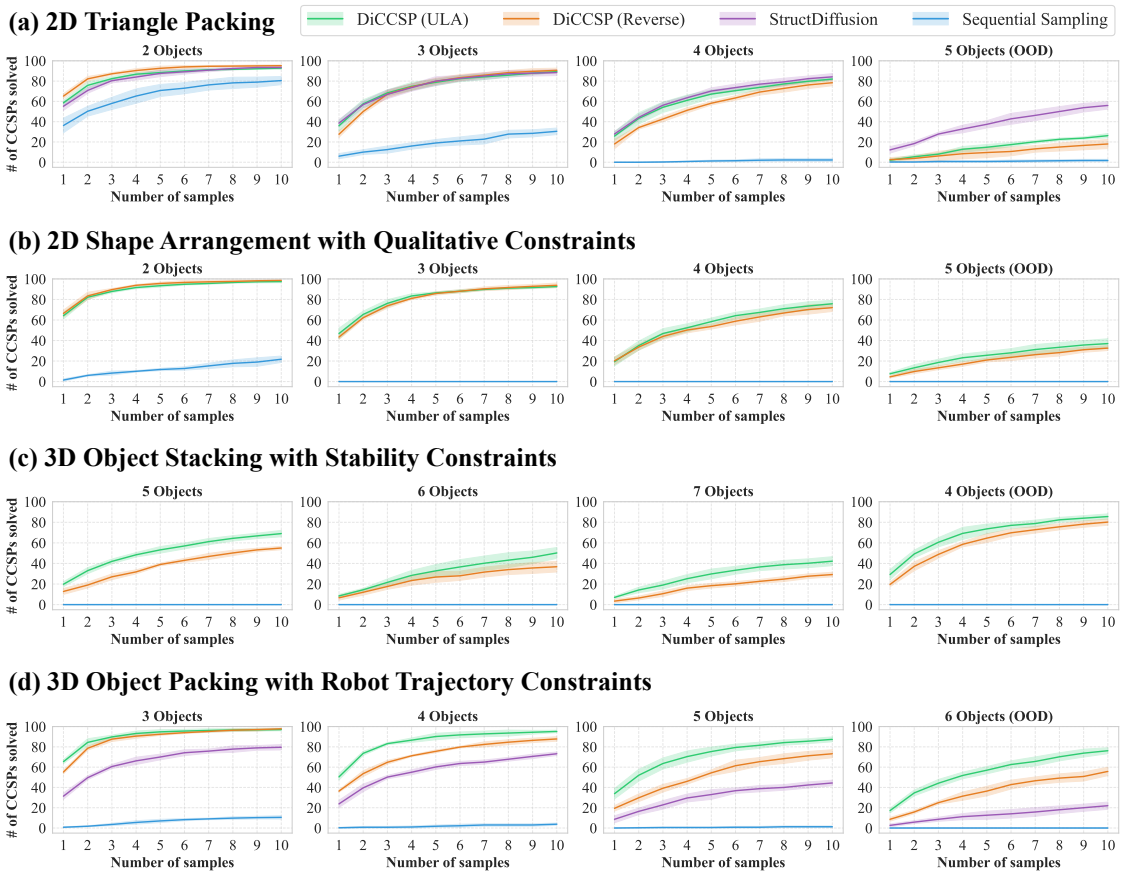


Figure 5-5: **Quantitative comparisons of constraint solvers.** Accumulated number of problems solved in 10 runs of different models. OOD=Out of training distribution. The shaded area indicates the standard deviation of various models across five different seeds. The sequential sampling baseline completely failed for task (c) and hard tasks in (b) and (d). Our full model performs better than a variant without ULA sampling and better than StructDiffusion in more complex problems (d).

objects, while the testing set contains 100 problems with 2 to 5 objects. Training examples are generated by randomly splitting the tray.

Fig. 5-5b shows the result, and Fig. 5-4b shows a concrete test problem that involves six objects and 45 constraints (including qualitative constraints shown in the figure and geometric constraints such as *in* and *cfree*). The task is very challenging when the number of objects and the number of constraints becomes large. Therefore, the baseline rejection sampling method barely solves any problem instances with 6 objects. Diffusion-CCSP significantly outperforms the baselines. We also visualized the learned energy landscape for a few constraints in Fig. 5-6 learned by our model, and more visualizations in Appendix E.6.

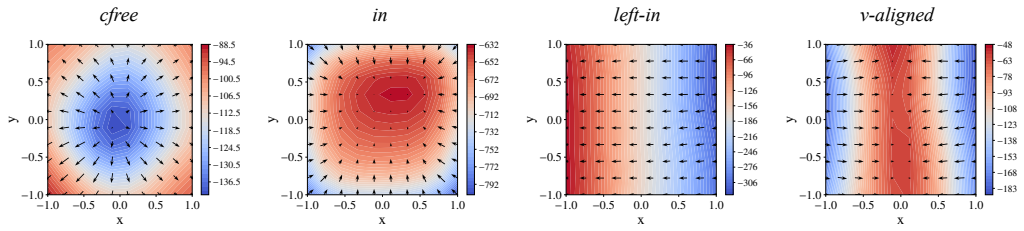


Figure 5-6: **Visualization of learned energy functions of selected geometric and qualitative constraints.**

5.3.3 3D Object Stacking with Stability Constraints

The third domain considers arranging trapezoidal prisms onto a shelf to satisfy a given support structure and stability. The goal is to generate a sequence of object placement actions (including object names and their target poses). This reflects real-life tasks such as organizing storage containers, which often involve tilted boxes due to varying prism heights. All of our training and testing problems involve at least one "bridge-like" structure, i.e., there is at least one object supported by multiple objects simultaneously. Furthermore, since the output of models is a sequence of object placements, a sequence is successful only if the final stage and *all intermediate states* are physically stable. We randomly sample the sizes of the shelf region and the shapes. We use a physical simulator to generate training examples. The training dataset has 24,000 problem instances and solutions with 5 to 7 objects, and the testing set includes 100 problem instances with 4 to 7 objects.

An example of the task is illustrated in Fig. 5-4c. Given that neither our algorithm nor the baselines specifically plan for object placement orders, we integrate them into a very simple task and motion planner, as detailed in Appendix E.5. This planner randomly samples object placement orders and box "support" structures (i.e., the arrangement of objects supporting other objects), then invokes the Diffusion-CCSP solver to find solutions. We assess two metrics: first, the performance of Diffusion-CCSP with a fixed CCSP graph, illustrated in Fig. 5-5c; second, the number of calls to Diffusion-CCSP required to solve problems involving varying quantities of objects, as shown in Table 5.1a. For instance, packing 6 boxes into a cabinet necessitates, on average, 25 different object placement and support structure samples. If a CCSP is not solvable by Diffusion-CCSP, it might truly be infeasible due to incorrect sampled placement orders or support structures. Therefore, Table 5.1a shows the integrated system's overall performance. Notably, given that we can process multiple CCSPs as a GPU batch and that generating object orders is inexpensive, the system does not have high latency overall. In a batch of 100 CCSPs, it takes on average 0.01-0.04 seconds for Diffusion-CCSP (Reverse) to solve each CCSP, and 0.06-0.19 seconds for Diffusion-CCSP (ULA).

Objects	N=4	N=5
# of Calls to Diffusion-CCSP	11.9	33.33

(a) 3D Object Stacking.

Objects	N=2	N=3	N=4	N=5	N=6
# of Calls to Diffusion-CCSP	1.33	1.97	2.96	4.37	7.26

(b) 3D Object Packing with Robots.

Table 5.1: **Average number of calls to Diffusion-CCSP in order to solve the full TAMP problem.**

5.3.4 Object Packing with Robot Trajectory Constraints

This domain involves packing 3D objects into a box container in a simulated environment using PyBullet, with a Franka Emika Panda arm. The dataset includes 53 objects from the PartNet Mobility dataset and ShapeNet [Chang *et al.*, 2015; Xiang *et al.*, 2020], 15 of which only afford side grasps, while others can be grasped from the top. We pre-generate a set of grasps for each object, and use a motion planner to find robot trajectories. The task and motion planner and Diffusion-CCSP jointly solve for grasping choices, object poses, and arm motions. Although our model doesn't directly generate grasps or trajectories, it reasons about object geometries and has learned to predict poses for which it is likely that there are collision-free trajectories (which is later verified with a motion planner.) The training dataset has 10,000 problem instances and solutions with 3 to 5 objects, and the testing set includes 100 problem instances with 3 to 6 objects. Diffusion-CCSP successfully finds solutions to 60-80 percent of the problems in just 10 samples for the most challenging 6-triangle packing problems.

Fig. 5-5d illustrates the number of samples needed for tasks involving a different number of objects. Overall, Diffusion-CCSP significantly outperforms the rejection-sampling-based solver and StructDiffusion, especially for larger numbers of objects. Diffusion-CCSP can even generalize directly to six-object packing problems without additional training. Table 5.1b showcases the full pipeline's performance, which includes generating placement orders, grasps, poses, and trajectories.

5.4 Related Work

Geometric rearrangement. A large body of work has studied geometric rearrangement of objects in robotics subject to conditions specified in natural language [Shridhar *et al.*, 2022a; Lynch *et al.*, 2020; Liu *et al.*, 2022b; Gkanatsios *et al.*, 2023; Mees *et al.*, 2022; Liu

et al., 2023b] or via formal specification or demonstration [Simeonov *et al.*, 2020; Curtis *et al.*, 2022; Paxton *et al.*, 2022; Yuan *et al.*, 2021; Bobu *et al.*, 2022; Simeonov *et al.*, 2022b]. Similar to our work, StructDiffusion [Liu *et al.*, 2023b] also uses diffusion models to solve for geometric rearrangements of objects. It employs a transformer-based diffusion model architecture to generate object placement poses that satisfy language-conditioned scene-level constraints, such as positioning all objects to form a circle. In comparison, our work provides a framework for solving arbitrary CCSPs that involve different constraints among objects.

Compositional generative modeling. Our work relates to existing work on compositional generative modeling, where individual generative models are combined to generate joint outputs [Du *et al.*, 2020; Liu *et al.*, 2021b; Nie *et al.*, 2021; Liu *et al.*, 2022a; Urain *et al.*, 2021; Du *et al.*, 2023; Wu *et al.*, 2022; Gkanatsios *et al.*, 2023], primarily in the setting of image generation. Most similar to our work, Gkanatsios *et al.* [2023] compose EBMs to represent different subgoals in compositions of object scenes. They currently only address object placements, whereas our method is generic and can combine constraints involving object poses, grasps, and robot configurations in the optimization-based sampling of object poses; this is generally more efficient than backtracking over values generated by a pose-only sampler.

Task and motion planning (TAMP). There are special-purpose 3D object packing algorithms based on heuristic-search or simulation [Egeblad *et al.*, 2007; Liu *et al.*, 2015; Wang and Hauser, 2019] for finding object poses that satisfy collision and stability constraints. They are again not directly comparable to Diffusion-CCSP, because they cannot address the same range and combinations of constraints.

5.5 Discussion

In conclusion, this chapter proposes a learning-based approach that leverages the compositionality of continuous constraint satisfaction problems (CCSP) and uses diffusion models as compositional constraint solvers. The proposed algorithm, namely the compositional diffusion constraint solver (Diffusion-CCSP), consists of modular diffusion models for different types of constraints. By conditioning these models on subsets of variables, solutions can be generated for the entire CCSP. The model exhibits strong generalization capabilities and can handle novel combinations of known constraints, even with more objects and constraints than encountered during training. It can also be integrated with search algorithms to solve general task and motion planning problems, which we will explore in Chapter 10 and

Chapter 11.

All constraints we have explored in this chapter have a fixed arity (i.e., they involve a fixed number of objects). An interesting direction is to consider variable-arity constraints. Another future improvement is to incorporate more sophisticated shape encoders and learning constraints from real-world data, such as online images, for a wider range of applications.

Part II

Neuro-Symbolic Visual Concepts

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Visual Concept Learning

We begin Part II by introducing the Neuro-Symbolic Concept Learner (NS-CL), a model that learns visual concepts, words, and semantic parsing of sentences without explicit supervision on any of them; instead, our model learns by simply looking at images and reading paired questions and answers. Our model builds an object-based scene representation and translates sentences into executable, symbolic programs. To bridge the learning of two modules, we use a neuro-symbolic reasoning module that executes these programs on the latent scene representation, an extension to the neuro-symbolic logic reasoning algorithms presented in Chapter 2. Analogous to human concept learning, the perception module learns visual concepts based on the language description of the object being referred to. Meanwhile, the learned visual concepts facilitate learning new words and parsing new sentences. We use curriculum learning to guide the search over the large compositional space of images and language. Extensive experiments demonstrate the accuracy and efficiency of our model on learning visual concepts, word representations, and the semantic parsing of sentences. Further, our method allows easy generalization to new object attributes, compositions, language concepts, scenes and questions, and even new program domains. It also empowers applications, including visual question answering and bidirectional image-text retrieval.

The material of this chapter has been previously published in Mao *et al.* [2019].

6.1 Introduction

Humans are capable of learning visual concepts by jointly understanding vision and language [Fazly *et al.*, 2010; Chrupała *et al.*, 2015; Gauthier *et al.*, 2018]. Consider the example shown in Fig. 6-1-I. Imagine that someone with no prior knowledge of colors is presented with the images of the red and green cubes, paired with the questions and answers. They can easily identify the difference in objects' visual appearance (in this case, color), and align it to the corresponding words in the questions and answers (`Red` and `Green`). Other object

attributes (e.g., shape) can be learned in a similar fashion. Starting from there, humans are able to inductively learn the correspondence between visual concepts and word semantics (e.g., spatial relations and referential expressions, Fig. 6-1-II), and unravel compositional logic from complex questions assisted by the learned visual concepts (Fig. 6-1-III, also see Abend *et al.* [2017]).

Motivated by this, we propose the Neuro-Symbolic Concept Learner (NS-CL), which jointly learns visual perception, words, and semantic language parsing from images and question-answer pairs. NS-CL has three modules: a neural-based perception module that extracts object-level representations from the scene, a visually-grounded semantic parser for translating questions into executable programs, and a symbolic program executor that reads out the perceptual representation of objects, classifies their attributes/relations, and executes the program to obtain an answer.

NS-CL learns from natural supervision (i.e., images and QA pairs), requiring no annotations on images or semantic programs for sentences. Instead, analogical to human concept learning, it learns via curriculum learning. NS-CL starts by learning representations/concepts of individual objects from short questions (e.g., What's the color of the cylinder?) on simple scenes ($\leq$3 objects). By doing so, it learns object-based concepts such as colors and shapes. NS-CL then learns relational concepts by leveraging these object-based concepts to interpret object referrals (e.g., Is there a box right of a cylinder?). The model iteratively adapts to more complex scenes and highly compositional questions.

NS-CL's modularized design enables interpretable, robust, and accurate visual reasoning: it achieves state-of-the-art performance on the CLEVR dataset [Johnson *et al.*, 2017a]. More importantly, it naturally learns disentangled visual and language concepts, enabling combinatorial generalization w.r.t. both visual scenes and semantic programs. In particular, we demonstrate four forms of generalization. First, NS-CL generalizes to scenes with more objects and longer semantic programs than those in the training set. Second, it generalizes to new visual attribute compositions, as demonstrated on the CLEVR-CoGenT [Johnson *et al.*, 2017a] dataset. Third, it enables fast adaptation to novel visual concepts, such as learning a new color. Finally, the learned visual concepts transfer to new tasks, such as image-caption retrieval, without any extra fine-tuning.

6.2 Related Work

Our model is related to research on joint learning of vision and natural language. In particular, there are many papers that learn visual concepts from descriptive languages, such as image-captioning or visually-grounded question-answer pairs [Kiros *et al.*, 2014; Shi *et al.*, 2018; Mao *et al.*, 2016; Vendrov *et al.*, 2016; Ganju *et al.*, 2017], dense language

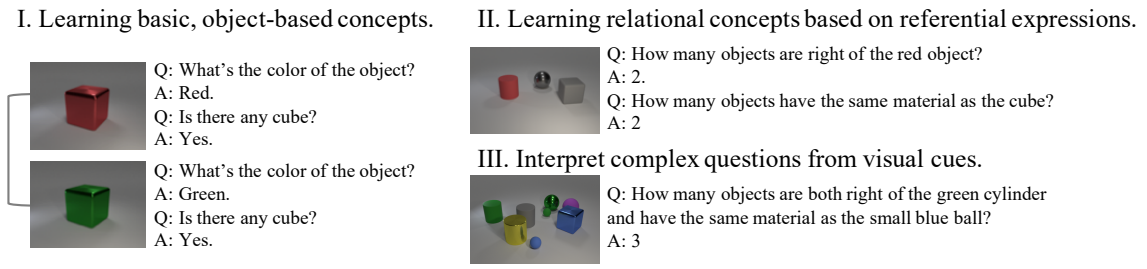


Figure 6-1: **Curriculum learning of visual concepts.** Humans learn visual concepts, words, and semantic parsing jointly and incrementally. **I.** Learning visual concepts (red *vs*. green) starts from looking at simple scenes, reading simple questions, and reasoning over contrastive examples [Fazly *et al.*, 2010]. **II.** Afterwards, we can interpret referential expressions based on the learned object-based concepts, and learn relational concepts (e.g., on the right of, the same material as). **III** Finally, we can interpret complex questions from visual cues by exploiting the compositional structure.

descriptions for scenes [Johnson *et al.*, 2016], video-captioning [Donahue *et al.*, 2015] and video-text alignment [Zhu *et al.*, 2015c].

Visual question answering (VQA) stands out as it requires understanding both visual content and language. The state-of-the-art approaches usually use neural attentions [Malinowski and Fritz, 2014; Chen *et al.*, 2015; Yang *et al.*, 2016; Xu and Saenko, 2016]. Beyond question answering, Johnson *et al.* [2017a] proposed the CLEVR (VQA) dataset to diagnose reasoning models. CLEVR contains synthetic visual scenes and questions generated from latent programs. Table 6.1 compares our model with state-of-the-art visual reasoning models [Andreas *et al.*, 2016a; Suarez *et al.*, 2018; Santoro *et al.*, 2017] along four directions: visual features, semantics, inference, and the requirement of extra labels.

For visual representations, Johnson *et al.* [2017b] encoded visual scenes into a convolutional feature map for program operators. Mascharka *et al.* [2018]; Hudson and Manning [2018] used attention as intermediate representations for transparent program execution. Recently, Yi *et al.* [2018] explored an interpretable, object-based visual representation for visual reasoning. It performs well, but requires fully-annotated scenes during training. Our model also adopts an object-based visual representation, but the representation is learned only based on natural supervision (questions and answers).

Anderson *et al.* [2018] also proposed to represent the image as a collection of convolutional object features and gained substantial improvements on VQA. Their model encodes questions with neural networks and answers the questions by question-conditioned attention over the object features. In contrast, NS-CL parses question inputs into programs and executes them on object features to get the answer. This makes the reasoning process interpretable and supports combinatorial generalization over quantities (e.g., counting objects). Our model also learns general visual concepts and their association with symbolic

Models	Visual Features	Semantics	Extra Labels		Inference
			# Prog.	Attr.	
FiLM [Perez *et al.*, 2018]	Convolutional	Implicit	0	No	Feature Manipulation
IEP [Johnson *et al.*, 2017b]	Convolutional	Explicit	700K	No	Feature Manipulation
MAC [Hudson and Manning, 2018]	Attentional	Implicit	0	No	Feature Manipulation
Stack-NMN [Hu *et al.*, 2018]	Attentional	Implicit	0	No	Attention Manipulation
TbD [Mascharka *et al.*, 2018]	Attentional	Explicit	700K	No	Attention Manipulation
NS-VQA [Yi *et al.*, 2018]	Object-Based	Explicit	0.2K	Yes	Symbolic Execution
NS-CL	Object-Based	Explicit	0	No	Symbolic Execution

Table 6.1: **Visual reasoning frameworks on the CLEVR dataset.** We compare NS-CL with other frameworks on the CLEVR VQA dataset, w.r.t. visual features, implicit or explicit semantics and supervisions.

representations of language. These learned concepts can then be explicitly interpreted and deployed in other vision-language applications such as image caption retrieval.

There are two types of approaches in semantic sentence parsing for visual reasoning: implicit programs as conditioned neural operations (e.g., conditioned convolution and dual attention) [Perez *et al.*, 2018; Hudson and Manning, 2018] and explicit programs as sequences of symbolic tokens [Andreas *et al.*, 2016a; Johnson *et al.*, 2017b; Mascharka *et al.*, 2018]. As a representative, Andreas *et al.* [2016a] build modular and structured neural architectures based on programs for answering questions. Explicit programs gain better interpretability, but usually require extra supervision such as ground-truth program annotations for training. This restricts their application. We propose to use visual grounding as distant supervision to parse questions in natural languages into explicit programs, with *zero* program annotations. Given the semantic parsing of questions into programs, Yi *et al.* [2018] proposed a purely symbolic executor for the inference of the answer in the logic space. Compared with theirs, we propose a quasi-symbolic executor for VQA.

Our work is also related to learning interpretable and disentangled representations for visual scenes using neural networks. Kulkarni *et al.* [2015] proposed convolutional inverse graphics networks for learning and inferring pose of faces, while Yang *et al.* [2015b] learned disentangled representation of pose of chairs from images. Wu *et al.* [2017] proposed the neural scene de-rendering framework as an inverse process of any rendering process. Siddharth *et al.* [2017]; Higgins *et al.* [2018] learned disentangled representations using deep generative models. In contrast, we propose an alternative representation learning approach through joint reasoning with language.

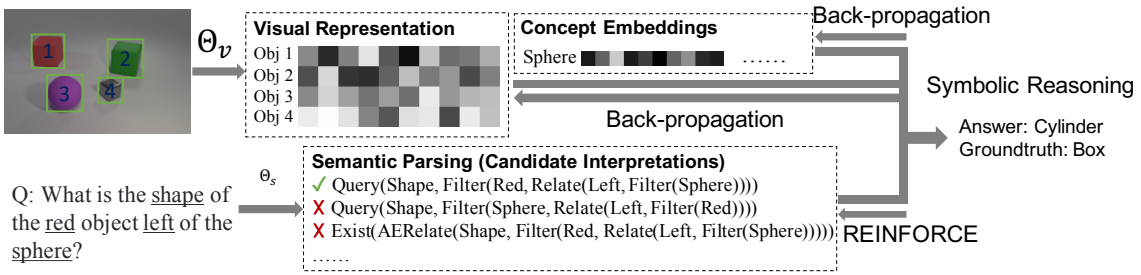


Figure 6-2: **Neuro-symbolic reasoning.** We propose to use neural symbolic reasoning as a bridge to jointly learn visual concepts, words, and semantic parsing of sentences.

6.3 Neuro-Symbolic Concept Learner

We present our Neuro-Symbolic Concept Learner, which uses a symbolic reasoning process to bridge the learning of visual concepts, words, and semantic parsing of sentences without explicit annotations for any of them. We first use a visual perception module to construct an object-based representation for a scene, and run a semantic parsing module to translate a question into an executable program. We then apply a quasi-symbolic program executor to infer the answer based on the scene representation. We use paired images, questions, and answers to jointly train the visual and language modules.

Shown in Fig. 6-2, given an input image, the visual perception module detects objects in the scene and extracts a deep, latent representation for each of them. The semantic parsing module translates an input question in natural language into an executable program given a domain specific language (DSL). The generated programs have a hierarchical structure of symbolic, functional modules, each fulfilling a specific operation over the scene representation. The explicit program semantics enjoys compositionality, interpretability, and generalizability.

The program executor executes the program upon the derived scene representation and answers the question. Our program executor works in a symbolic and deterministic manner. This feature ensures a transparent execution trace of the program. Our program executor has a fully differentiable design w.r.t. the visual representations and the concept representations, which supports gradient-based optimization during training.

6.3.1 Model details

Visual perception. Shown in Fig. 6-2, given the input image, we use a pretrained Mask R-CNN [He *et al.*, 2017] to generate object proposals for all objects. The bounding box for each single object paired with the original image is then sent to a ResNet-34 [He *et al.*, 2015] to extract the region-based (by RoI Align) and image-based features respectively. We

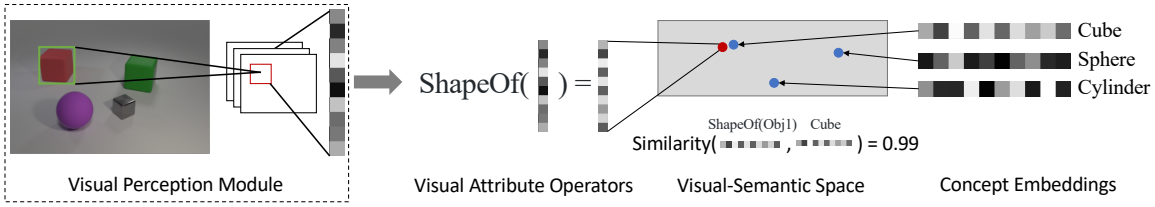


Figure 6-3: **Grounding of visual concepts.** We treat attributes such as `Shape` and `Color` as neural operators. The operators map object representations into a visual-semantic space. We use similarity-based metric to classify objects.

concatenate them to represent each object. Here, the inclusion of the representation of the full scene adds the contextual information, which is essential for the inference of relative attributes such as size or spatial position.

Concept quantization. Visual reasoning requires determining an object's attributes (e.g., its color or shape). We assume each visual attribute (e.g., shape) contains a set of visual concept (e.g., `Cube`). In NS-CL, visual attributes are implemented as neural operators , mapping the object representation into an attribute-specific embedding space. Fig. 6-3 shows an inference an object's shape. Visual concepts that belong to the shape attribute, including `Cube`, `Sphere` and `Cylinder`, are represented as vectors in the shape embedding space. These concept vectors are also learned along the process. We measure the cosine distances $\langle \cdot, \cdot \rangle$ between these vectors to determine the shape of the object. Specifically, we compute the probability that an object o_i is a cube by $\sigma\Big(\langle \text{ShapeOf}(o_i), v^{\texttt{Cube}} \rangle - \gamma\Big)/\tau$, where ShapeOf$(\cdot)$ denotes the neural operator, $v^{\texttt{Cube}}$ the concept embedding of `Cube` and σ the Sigmoid function. γ and τ are scalar constants for scaling and shifting the values of similarities. We classify relational concepts (e.g., `Left`) between a pair of objects similarly, except that we concatenate the visual representations for both objects to form the representation of their relation.

DSL and semantic parsing. The semantic parsing module translates a natural language question into an executable program with a hierarchy of primitive operations, represented in a domain-specific language (DSL) designed for VQA. The DSL covers a set of fundamental operations for visual reasoning, such as filtering out objects with certain concepts or query the attribute of an object. The operations share the same input and output interface, and thus can be compositionally combined to form programs of any complexity. We include a complete specification of the DSL used by our framework in the Appendix F.1.

Our semantic parser generates the hierarchies of latent programs in a sequence to tree manner [Dong and Lapata, 2016]. We use a bidirectional GRU [Cho *et al.*, 2014] to encode an input question, which outputs a fixed-length embedding of the question. A decoder based

on GRU cells is applied to the embedding, and recovers the hierarchy of operations as the latent program. Some operations takes concepts their parameters, such as `Filter(` Red `)` and `Query(` Shape `)`. These concepts are chosen from all concepts appeared in the input question. Fig. 6-4(B) shows an example, while more details can be found in Appendix F.2.

Quasi-symbolic program execution. Given the latent program recovered from the question in natural language, a symbolic program executor executes the program and derives the answer based on the object-based visual representation. Our program executor is a collection of deterministic functional modules designed to realize all logic operations specified in the DSL. Fig. 6-4(B) shows an illustrative execution trace of a program. These techniques are built on top of ideas from Chapter 2.

To make the execution differentiable w.r.t. visual representations, we represent the intermediate results in a probabilistic manner: a set of objects is represented by a vector, as the attention mask over all objects in the scene. Each element, $\text{Mask}_i \in [0, 1]$ denotes the probability that the i-th object of the scene belongs to the set. For example, shown in Fig. 6-4(B), the first `Filter` operation outputs a mask of length 4 (there are in total 4 objects in the scene), with each element representing the probability that the corresponding object is selected out (i.e., the probability that each object is a green cube). The output "mask" on the objects will be fed into the next module (`Relate` in this case) as input and the execution of programs continues. The last module outputs the final answer to the question. We refer interested readers to Appendix F.3 for the implementation of all operators.

6.3.2 Training paradigm

Optimization objective. The optimization objective of NS-CL is composed of two parts: concept learning and language understanding. Our goal is to find the optimal parameters Θ_v of the visual perception module Perception (including the ResNet-34 for extracting object features, attribute operators. and concept embeddings) and Θ_s of the semantic parsing module SemanticParse, to maximize the likelihood of answering the question Q correctly:

$$\Theta_v, \Theta_s \leftarrow \arg\max_{\Theta_v, \Theta_s} \mathbb{E}_P[\Pr[A = \text{Executor}(\text{Perception}(S; \Theta_v), P)]], \tag{6.1}$$

where P denotes the program, A the answer, S the scene, and Executor the quasi-symbolic executor. The expectation is taken over $P \sim \text{SemanticParse}(Q; \Theta_s)$.

Recall the program executor is fully differentiable w.r.t. the visual representation. We compute the gradient w.r.t. Θ_v as $\nabla_{\Theta_v} \mathbb{E}_P[D_{\text{KL}}(\text{Executor}(\text{Perception}(S; \Theta_v), P) \| A)]$. We use REINFORCE [Williams, 1992] to optimize the semantic parser Θ_s: $\nabla_{\Theta_s} = \mathbb{E}_P[r \cdot \log \Pr[P = \text{SemanticParse}(Q; \Theta_s)]]$, where the reward $r = 1$ if the answer is correct and 0 otherwise.

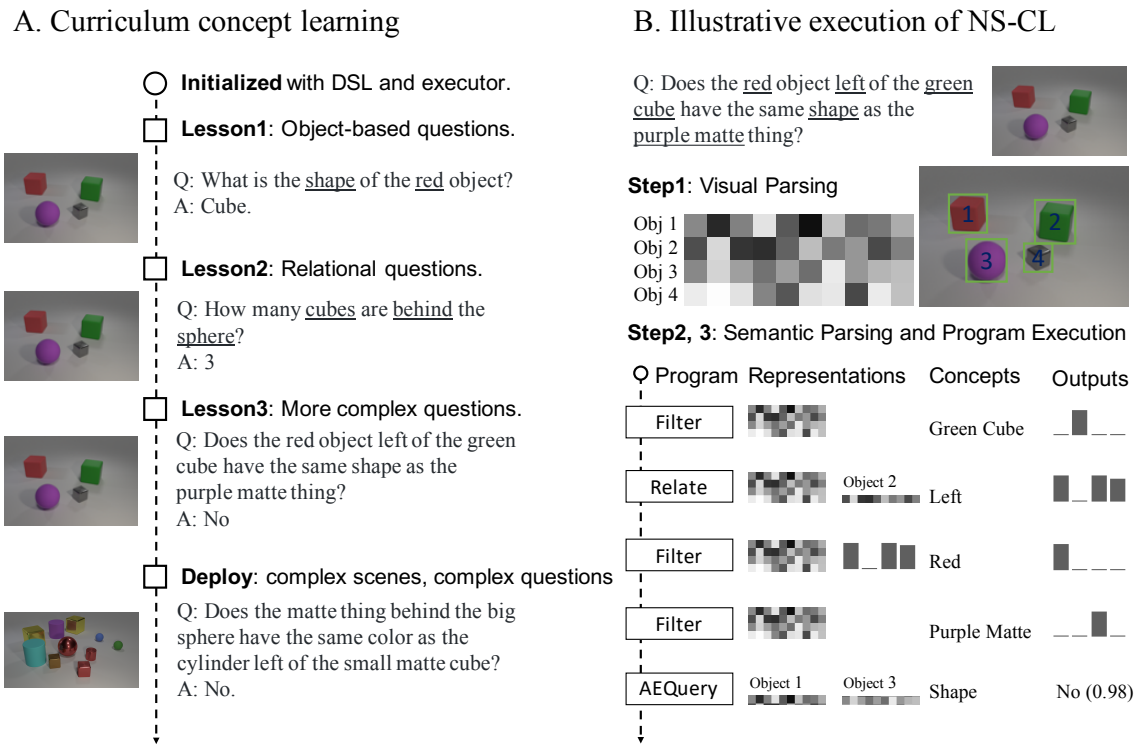


Figure 6-4: **Curriculum learning and illustrative execution of NS-CL. A.** Demonstration of the curriculum learning of visual concepts, words, and semantic parsing of sentences by watching images and reading paired questions and answers. Scenes and questions of different complexities are illustrated to the learner in an incremental manner. **B.** Illustration of our neuro-symbolic inference model for VQA. The perception module begins with parsing visual scenes into object-based deep representations, while the semantic parser parse sentences into executable programs. A symbolic execution process bridges two modules.

We also use off-policy search to reduce the variance of REINFORCE, the detail of which can be found in Appendix F.4.

Curriculum visual concept learning. Motivated by human concept learning as in Fig. 6-1, we employ a curriculum learning approach to help joint optimization. We heuristically split the training samples into four stages (Fig. 6-4(A)): first, learning object-level visual concepts; second, learning relational questions; third, learning more complex questions with perception modules fixed; fourth, joint fine-tuning of all modules. We found that this is essential to the learning of our Neuro-Symbolic Concept Learner. We include more technical details in Appendix F.5.

6.4 Experiments

We demonstrate the following advantages of our NS-CL. First, it learns visual concepts with remarkable accuracy; second, it allows data-efficient visual reasoning on the CLEVR dataset [Johnson *et al.*, 2017a]; third, it generalizes well to new attributes, visual composition, and language domains.

We train NS-CL on 5K images (<10% of CLEVR's 70K training images). We generate 20 questions for each image for the entire curriculum learning process. The Mask R-CNN module is pretrained on 4K generated CLEVR images with bounding box annotations, following Yi *et al.* [2018].

6.4.1 Visual concept learning

Classification-based concept evaluation. Our model treats attributes as neural operators that map latent object representations into an attribute-specific embedding space (Fig. 6-3). We evaluate the concept quantization of objects in the CLEVR validation split. Our model can achieve near perfect classification accuracy (~99%) for all object properties, suggesting it effectively learns generic concept representations. The result for spatial relations is relatively lower, because CLEVR does not have direct queries on the spatial relation between objects. Thus, spatial relation concepts can only be learned indirectly.

Count-based concept evaluation. The SOTA methods do not provide interpretable representation on individual objects [Johnson *et al.*, 2017a; Hudson and Manning, 2018; Mascharka *et al.*, 2018] . To evaluate the visual concepts learned by such models, we generate a synthetic question set. The diagnostic question set contains simple questions as the following form: "How many `red` objects are there?". We evaluate the performance on all concepts appeared in the CLEVR dataset.

Table 6.2 summarizes the results compared with strong baselines, including methods based on convolutional features [Johnson *et al.*, 2017b] and those based on neural attentions [Mascharka *et al.*, 2018; Hudson and Manning, 2018]. Our approach outperforms IEP by a significant margin (8%) and attention-based baselines by >2%, suggesting object-based visual representations and symbolic reasoning helps to interpret visual concepts.

6.4.2 Data-efficient and interpretable visual reasoning

NS-CL jointly learns visual concepts, words and semantic parsing by watching images and reading paired questions and answers. It can be directly applied to VQA.

	Visual	Mean	Color	Mat.	Shape	Size
IEP	Conv.	90.6	91.0	90.0	89.9	90.6
MAC	Attn.	95.9	98.0	91.4	94.4	94.2
TbD (hres.)	Attn.	96.5	96.6	92.2	95.4	92.6
NS-CL	Obj.	**98.7**	**99.0**	**98.7**	**98.1**	**99.1**

Table 6.2: We also evaluate the learned visual concepts using a diagnostic question set containing simple questions such as "How many `red` objects are there?". NS-CL outperforms both convolutional and attentional baselines. The suggested object-based visual representation and symbolic reasoning approach perceives better interpretation of visual concepts.

Model	Visual	Accuracy (100% Data)	Accuracy (10% Data)
TbD	Attn.	99.1	54.2
TbD-Object	Obj.	84.1	52.6
TbD-Mask	Attn.	99.0	55.0
MAC	Attn.	98.9	67.3
MAC-Object	Obj.	79.5	51.2
MAC-Mask	Attn.	98.7	68.4
NS-CL	Obj.	**99.2**	**98.9**

Table 6.3: We compare different variants of baselines for a systematic study on visual features and data efficiency. Using only 10% of the training images, our model is able to achieve a comparable results with the baselines trained on the full dataset. See the text for details.

Table 6.4 summarizes results on the CLEVR validation split. Our model achieves the state-of-the-art performance among all baselines using zero program annotations, including MAC [Hudson and Manning, 2018] and FiLM [Perez *et al.*, 2018]. Our model achieves comparable performance with the strong baseline TbD-Nets [Mascharka *et al.*, 2018], whose semantic parser is trained using 700K programs in CLEVR (ours need 0). The recent NS-VQA model from Yi *et al.* [2018] achieves better performance on CLEVR; however, their system requires annotated visual attributes and program traces during training, while our NS-CL needs no extra labels.

Here, the visual perception module is pre-trained on ImageNet [Deng *et al.*, 2009]. Without pre-training, the concept learning accuracies drop by 0.2% on average and the QA accuracy drops by 0.5%. Meanwhile, NS-CL recovers the underlying programs of questions accurately (> 99.9% accuracy). NS-CL can also detect ambiguous or invalid programs and indicate exceptions. Please see Appendix F.6 for more details. NS-CL can also be applied to other visual reasoning testbeds. Please refer to Appendix F.7.1 for our results on the Minecraft dataset [Yi *et al.*, 2018].

For a systematic study on visual features and data efficiency, we implement two variants of the baseline models: TbD-Object and MAC-Object. Inspired by [Anderson *et al.*, 2018], instead of the input image, TbD-Object and MAC-Object take a stack of object features as input. TbD-Mask and MAC-Mask integrate the masks of objects by using them to guide the attention over the images.

Table 6.3 summarizes the results. Our model outperforms all baselines on data efficiency. This comes from the full disentanglement of visual concept learning and symbolic reasoning: how to execute program instructions based on the learned concepts is programmed. TbD-

Model	Prog. Anno.	Overall	Count	Cmp. Num.	Exist	Query Attr.	Cmp. Attr.
Human	N/A	92.6	86.7	86.4	96.6	95.0	96.0
NMN	700K	72.1	52.5	72.7	79.3	79.0	78.0
N2NMN	700K	88.8	68.5	84.9	85.7	90.0	88.8
IEP	700K	96.9	92.7	98.7	97.1	98.1	98.9
DDRprog	700K	98.3	96.5	98.4	98.8	99.1	99.0
TbD	700K	**99.1**	**97.6**	**99.4**	**99.2**	**99.5**	**99.6**
RN	0	95.5	90.1	93.6	97.8	97.1	97.9
FiLM	0	97.6	94.5	93.8	99.2	99.2	99.0
MAC	0	98.9	97.2	**99.4**	**99.5**	99.3	**99.5**
NS-CL	0	**98.9**	**98.2**	99.0	98.8	**99.3**	99.1

Table 6.4: Our model outperforms all baselines using no program annotations. It achieves comparable results with models trained by full program annotations such as TbD.

Model	Test			
	Split A	Split B	Split C	Split D
MAC	97.3	N/A	92.9	N/A
IEP	96.1	92.1	91.5	90.9
TbD	98.8	94.5	94.3	91.9
NS-CL	**98.9**	**98.9**	**98.7**	**98.8**

Figure 6-5: We test the combinatorial generalization w.r.t. the number of objects in scenes and the complexity of questions (i.e. the depth of the program trees). We makes four split of the data containing various complexities of scenes and questions. Our object-based visual representation and explicit program semantics enjoys the best (and almost-perfect) combinatorial generalization compared with strong baselines.

Object and MAC-Object demonstrate inferior results in our experiments. We attribute this to the design of model architectures and have a detailed analysis in Appendix F.6.3. Although TbD-Mask and MAC-Mask do not perform better than the originals, we find that using masks to guide attentions speeds up the training.

Besides achieving a competitive performance on the visual reasoning testbeds, by leveraging both object-based representation and symbolic reasoning, out model learns fully interpretable visual concepts: see Appendix F.8 for qualitative results on various datasets.

6.4.3 Generalization to new attributes and compositions

Generalizing to new visual compositions. The CLEVR-CoGenT dataset is designed to evaluate models' ability to generalize to new visual compositions. It has two splits: Split A only contains gray, blue, brown and yellow cubes, but red, green, purple, and cyan cylinders; split B imposes the opposite color constraints on cubes and cylinders. If we directly learn visual concepts on split A, it overfits to classify shapes based on the color, leading to a poor generalization to split B.

Our solution is based on the idea of seeing attributes as operators. Specifically, we jointly train the concept embeddings (e.g., `Red`, `Cube`, etc.) as well as the semantic parser on split A, keeping pre-trained, frozen attribute operators. As we learn distinct representation spaces for different attributes, our model achieves an accuracy of 98.8% on split A and 98.9% on

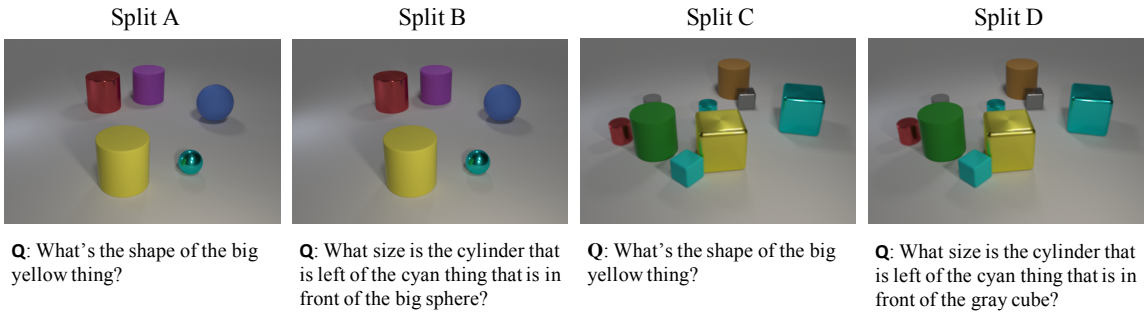


Figure 6-6: Samples collected from four splits in Section 6.4.3 for illustration. Models are trained on split A but evaluated on all splits for testing the combinatorial generalization.

split B.

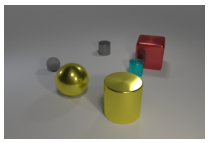
Generalizing to new visual concepts. We expect the process of concept learning can take place in an incremental manner: having learned 7 different colors, humans can learn the 8-th color incrementally and efficiently. To this end, we build a synthetic split of the CLEVR dataset to replicate the setting of incremental concept learning. Split A contains only images without any purple objects, while split B contains images with at least one purple object. We train all the models on split A first, and finetune them on 100 images from split B. We report the final QA performance on split B's validation set. All models use a pre-trained semantic parser on the full CLEVR dataset.

Our model performs a 93.9% accuracy on the QA test in Split B, outperforming the convolutional baseline IEP [Johnson *et al.*, 2017b] and the attentional baseline TbD [Mascharka *et al.*, 2018] by 4.6% and 6.1% respectively. The acquisition of `Color` operator brings more efficient learning of new visual concepts.

6.4.4 Combinatorial generalization to new scenes and questions

Having learned visual concepts on small-scale scenes (containing only few objects) and simple questions (only single-hop questions), we humans can easily generalize the knowledge to larger-scale scenes and to answer complex questions. To evaluate this, we split the CLEVR dataset into four parts: **Split A** contains only scenes with less than 6 objects, and questions whose latent programs having a depth less than 5; **Split B** contains scenes with less than 6 objects, but arbitrary questions; **Split C** contains arbitrary scenes, but restricts the program depth being less than 5; **Split D** contains arbitrary scenes and questions. Figure 6-6 shows some illustrative samples.

As VQA baselines are unable to count a set of objects of arbitrary size, for a fair comparison, all programs containing the "count" operation over > 6 objects are removed



Caption: There is a big yellow cylinder in front of a gray object.

(a) An illustrative pair of image and caption in our synthetic dataset.

Model	Retrieval Acc.
IEP	95.5
TbD	**97.0**
NS-CL	96.9

(b) Image-caption retrieval accuracy on a subset of data. Our model archives comparable results with VQA baselines.

Model	Retrieval Acc.
CNN-LSTM	68.9
NS-CL	**97.0**

(c) Image-caption retrieval accuracy on the full dataset. Our model outperforms baselines and requires no extra training or fine-tuning of the visual perception module.

Table 6.5: We introduce a new simple DSL for image-caption retrieval to evaluate how well the learned visual concepts transfer. Due to the difference between VQA and caption retrieval, VQA baselines are only able to infer the result on a partial set of data. The learned object-based visual concepts can be directly transferred into the new domain for free.

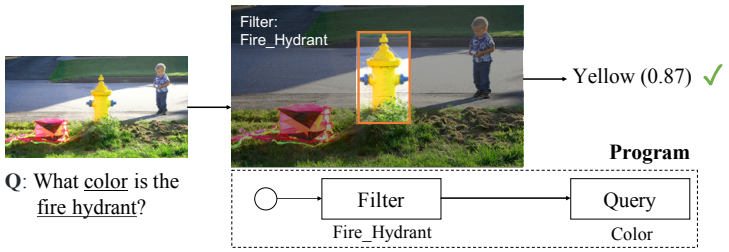
from the set. For methods using explicit program semantics, the semantic parser is pre-trained on the full dataset and fixed. Methods with implicit program semantics [Hudson and Manning, 2018] learn an entangled representation for perception and reasoning, and cannot trivially generalize to more complex programs. We only use the training data from the Split A and then quantify the generalization ability on other three splits. Shown in Table 6-5, our NS-CL leads to almost-perfect generalization to larger scenes and more complex questions, outperforming all baselines by at least 4% in QA accuracy.

6.4.5 Extending to other program domain

The learned visual concepts can also be used in other domains such as image retrieval. With the visual scenes fixed, the learned visual concepts can be directly transferred into the new domain. We only need to learn the semantic parsing of natural language into the new DSL.

We build a synthetic dataset for image retrieval and adopt a DSL from scene graph–based image retrieval [Johnson *et al.*, 2015]. The dataset contains only simple captions: "There is an <object A> <relation> <object B>." (e.g., There is a box right of a cylinder). The semantic parser learns to extract corresponding visual concepts (e.g., `box`, `right`, and `cylinder`) from the sentence. The program can then be executed on the visual representation to determine if the visual scene contains such relational triples.

For simplicity, we treat retrieval as classifying whether a relational triple exists in the image. This functionality cannot be directly implemented on the CLEVR VQA program domain, because questions such as "Is there a box right of a cylinder" can be ambiguous if there exist multiple cylinders in the scene. Due to the entanglement of the visual representation with the specific DSL, baselines trained on CLEVR QA can not be directly applied to this task. For a fair comparison with them, we show the result in Table 6.5b



Model	Accuracy
MLP	43.9
MAC	**46.2**
NS-CL	44.3

Figure 6-7: **Left**: An example image-question pair from the VQS dataset and the corresponding execution trace of NS-CL. **Right**: Results on the VQS test set. Our model achieves a comparable results with the baselines.

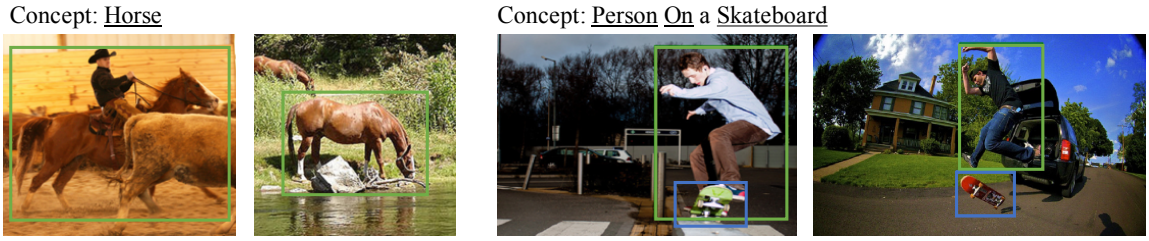


Figure 6-8: **Concepts learned from VQS**, including object categories, attributes, and relations.

on a subset of the generated image-caption pairs where the underlying programs have no ambiguity regarding the reference of object B. A separate semantic parser is trained for the VQA baselines, which translates captions into a CLEVR QA-compatible program (e.g., `Exist(Filter(Box, Relate(Right, Filter(Cylinder))))`.

Table 6.5c compares our NS-CL against typical image-text retrieval baselines on the full image-caption dataset. Without any annotations of the sentence semantics, our model learns to parse the captions into the programs in the new DSL. It outperforms the CNN-LSTM baseline by 30%.

6.4.6 Extending to natural images and language

We further conduct experiments on MS-COCO [Lin *et al.*, 2014b] images. Results are presented on the VQS dataset [Gan *et al.*, 2017]. VQS contains a subset of images and questions from the original VQA 1.0 dataset [Antol *et al.*, 2015]. All questions in the VQS dataset can be visually grounded: each question is associated with multiple image regions, annotated by humans as essential for answering the question. Fig. 6-7 illustrates an execution trace of NS-CL on VQS.

We use a syntactic dependency parser to extract programs and concepts from language [Andreas *et al.*, 2016a; Schuster *et al.*, 2015]. The object proposals and features are extracted from models pre-trained on the MS-COCO dataset and the ImageNet dataset,

respectively. Illustrated in Fig. 6-7, our model shows competitive performance on QA accuracy, comparable with the MLP baseline [Jabri *et al.*, 2016] and the MAC network [Hudson and Manning, 2018]. Additional illustrative execution traces of NS-CL are in Appendix F.8. Beyond answering questions, NS-CL effectively learns visual concepts from data. Fig. 6-8 shows examples of the learned visual concepts, including object categories, attributes, and relations. Experiment setup and implementation details are in Appendix F.7.2.

In this chapter, we focus on a neuro-symbolic framework that learns visual concepts about object properties and relations. Indeed, visual question answering requires AI systems to reason about more general concepts such as events or activities [Levin, 1993]. We leave the extension of NS-CL along this direction and its application to general VQA datasets [Antol *et al.*, 2015] as future work.

6.5 Discussion

We presented a method that jointly learns visual concepts, words, and semantic parsing of sentences from natural supervision. The proposed framework, NS-CL, learns by looking at images and reading paired questions and answers, without any explicit supervision such as class labels for objects. Our model learns visual concepts with remarkable accuracy. Based upon the learned concepts, our model achieves good results on question answering, and more importantly, generalizes well to new visual compositions, new visual concepts, and new domain specific languages.

Of course, such improvements come at a cost: the neuro-symbolic concept learning frameworks have the limitation of relying on a predefined DSL. This DSL encompasses primitive operators such as `Filter`, as well as concept symbols such as *orange* and *place*. In the following, we are going to delve into these two parts.

In most domains, by combining object property primitives, relational primitives, and action primitives, along with simple set operations such as intersection, union, and counting, we can construct a highly capable system. Recently, there has been a growing interest in extending the primitive set and number operations to general programming languages, such as Python code [Dídac *et al.*, 2023]. This will significantly improve the expressiveness of the programs by including complex control flows such as loops and recursions. However, in general, this also introduces new challenges in the learning of concepts. In NS-CL, the concept representations are learned through back-propagation of the program execution trace. Although there has been much work on backpropagation against program execution traces [Petersen *et al.*, 2021], complex control flows will inevitably introduce intractability of possible execution results, as well as gradient vanishing and explosion problems.

Relying on a predefined set of concepts can be too restrictive in many real-world

applications. In NS-CL, we have assumed the knowledge of a set of concept words in the dataset. We will discuss two approaches to address this. First, in Chapter 8, we will leverage large language models that have been trained on Internet-scale text and code corpora to translate natural language queries into programs with concept symbols. These concept symbols are not chosen from a given vocabulary, but are instead automatically generated by LLMs based on the user queries. For each concept symbol that appears in LLM-translated programs, a new concept representation will be initialized and learned.

Another approach that we will explore in Chapter 9, without the assumption of pre-training large language models, is grammar-based lexicon learning. The high-level idea behind grammar-based lexicon learning is that instead of trying to associate each word or phrase with a predefined set of concept names (i.e., mapping the word "orange" to a concept "ORANGE"), we construct the library of concepts by "converting" each word into a concept. This is roughly equivalent to discovering the syntax of each word. For example, if we see the word "orange" in a sentence and it is an adjective, then we immediately know the word "orange" should correspond to an object property concept, named "ORANGE." This approach is called "grammar-based lexicon learning" because the system begins with a small set of universal grammar rules and jointly discovers new concepts from the text corpus while learning their grounding.

In later chapters, we will also extend our framework to other domains such as 3D scene understanding and video understanding (Chapter 8) and robotic manipulation (Chapter 11). In those chapters, we will discover semantic representations for actions and interactions (e.g., push) beyond static spatial relations.

Chapter 7

Fast Visual Concept Learning with Metaconcepts

This chapter introduces *metaconcepts*, which are concepts describing relations between concepts, into the concept learning framework. We present a meta-learning framework for learning new visual concepts quickly, from just one or a few examples, guided by multiple naturally occurring data streams: simultaneously looking at images, reading sentences that describe the objects in the scene, and interpreting supplemental sentences that relate the novel concept with other concepts. The learned concepts support downstream applications, such as answering questions by reasoning about unseen images. Our model, namely FALCON, represents individual visual concepts, such as colors and shapes, as axis-aligned boxes in a high-dimensional space (the "*box embedding space*"). Given an input image and its paired sentence, our model first resolves the referential expression in the sentence and associates the novel concept with particular objects in the scene. Next, our model interprets supplemental sentences to relate the novel concept with other known concepts, such as "*X has property Y*" or "*X is a kind of Y*". Finally, it infers an optimal box embedding for the novel concept that jointly 1) maximizes the likelihood of the observed instances in the image, and 2) satisfies the relationships between the novel concepts and the known ones. We demonstrate the effectiveness of our model on both synthetic and real-world datasets.

The material of this chapter has been previously published in Mei *et al.* [2022]. Lingjie (Jerry) Mei, in particular, contributed significantly to the materials presented in this chapter.

7.1 Introduction

Humans build a cumulative knowledge repository of visual concepts throughout their lives from a diverse set of inputs: by looking at images, reading sentences that describe the

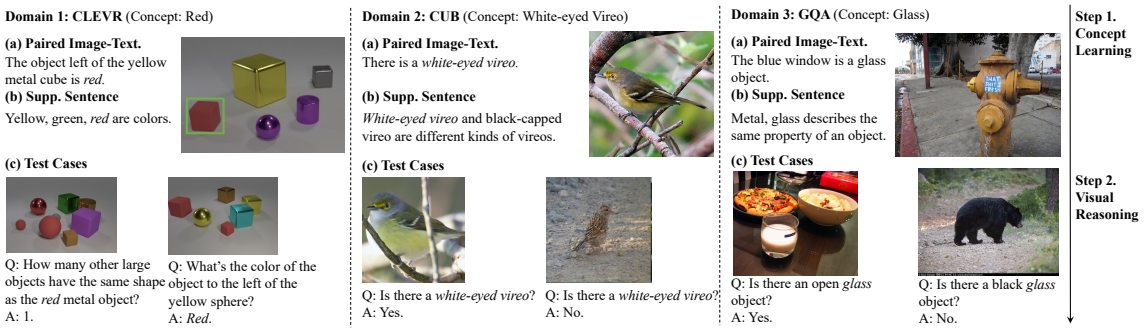


Figure 7-1: **Three illustrative examples of our fast concept learning task.** Our model learns from three naturally occurring data streams: (a) looking at images, reading sentences that describe the objects in the scene, and (b) interpreting supplemental sentences that relate the novel concept with other concepts. (c) The acquired novel concepts (e.g., *red* and *white-eyed vireo* transfer to downstream tasks, such as visual reasoning. We present a meta-learning framework to solve this task.

properties of the concept, etc. Importantly, adding a novel concept to the knowledge repository requires only a small amount of data, such as a few images about the concept and a short textual description [Bloom, 2000; Swingley, Daniel, 2010; Carey and Bartlett, 1978]. Take Fig. 7-1 as an example: from just a single image that contains many objects (Fig. 7-1a), as well as a short descriptive sentence that describes a new concept *red*: "the object left of the yellow cube is red", humans can effortlessly ground the novel word "red" with the visual appearance of the object being referred to [Bloom, 2000]. Supplemental sentences such as "red is a kind of color" (Fig. 7-1b) may provide additional information: "red" objects should be classified based on their hue. This further supports us to generalize the learned concept "red" to objects of various shapes, sizes, and materials. Finally, the acquired concept can be used flexibly in other tasks such as question answering (Fig. 7-1c).

Our goal is to build machines that can learn concepts that are associated with the physical world in an incremental manner and flexibly use them to answer queries. To learn a new concept, for example, the word *red* in Fig. 7-1a, the system should 1) interpret the semantics of the descriptive sentence composed of other concepts, such as *left*, *yellow*, and *cube*, 2) instantiate a representation for the novel concept with the visual appearance of the referred object, 3) mediate the concept representation based on the supplemental sentences that describe the property of the concept or relate it with other concepts, and 4) use the learned concept flexibly in different scenarios. A framework that can solve these challenges will allow us to build machines that can better learn from human and communicate with human.

To address these challenges, in this chapter, we present a unified framework, FALCON (FAst Learning of novel visual CONcepts). FALCON maintains a collection of embedding vectors for individual visual concepts, which naturally grows in an incremental way as it

learns more concepts. A neuro-symbolic concept learning and reasoning framework learns new concepts by looking at images and reading paired sentences, and use them to answer incoming queries.

Concretely, FALCON represents individual visual concepts, such as colors and shapes, as axis-aligned boxes in a high-dimensional space (the "*box embedding space*" [Vilnis *et al.*, 2018]), while objects in different images will be embedded into the same latent space as points. We say object X has property Y if the embedding vector X is inside the embedding box of Y. Given an input image and its paired sentence, our model first resolves the referential expression in the sentence using the previously seen concepts (e.g., *left*, *yellow*, and *cube*) and associate the novel concept with particular objects in the scene. Next, our model interprets supplemental sentences to relate the novel concept with other known concepts (e.g., *yellow*). To infer the box embedding of the novel concept, we train a neural network to predict the optimal box embedding for the novel concept that jointly 1) maximizes the data likelihood of the observed examples, and 2) satisfies the relationships between the novel concepts and the known ones. This module is trained with a meta-learning procedure.

This chapter makes the following contributions. First, we present a unified neuro-symbolic framework for fast visual concept learning from diverse data streams. Second, we introduce a new concept embedding prediction module that learns to integrate visual examples and conceptual relations to infer a novel concept embedding. Finally, we build a protocol for generating meta-learning test cases for evaluating fast visual concept learning, by augmenting existing visual reasoning datasets and knowledge graphs. By evaluation on both synthetic and natural image datasets, we show that our model learns more accurate representations for novel concepts compared with existing baselines for fast concept learning. Systematical studies also show that our model can efficiently use the supplemental concept descriptions to resolve ambiguities in the visual examples. We also provide discussions about the design of different modules in our system.

7.2 FALCON

The proposed model, FALCON, learns visual concepts by simultaneously looking at images, reading sentences that describe the objects in the scene, and interpreting supplemental sentences that describe the properties of the novel concepts. FALCON learns to learn novel concepts quickly and in a continual manner. We start with a formal definition of our fast and continual concept learning task.

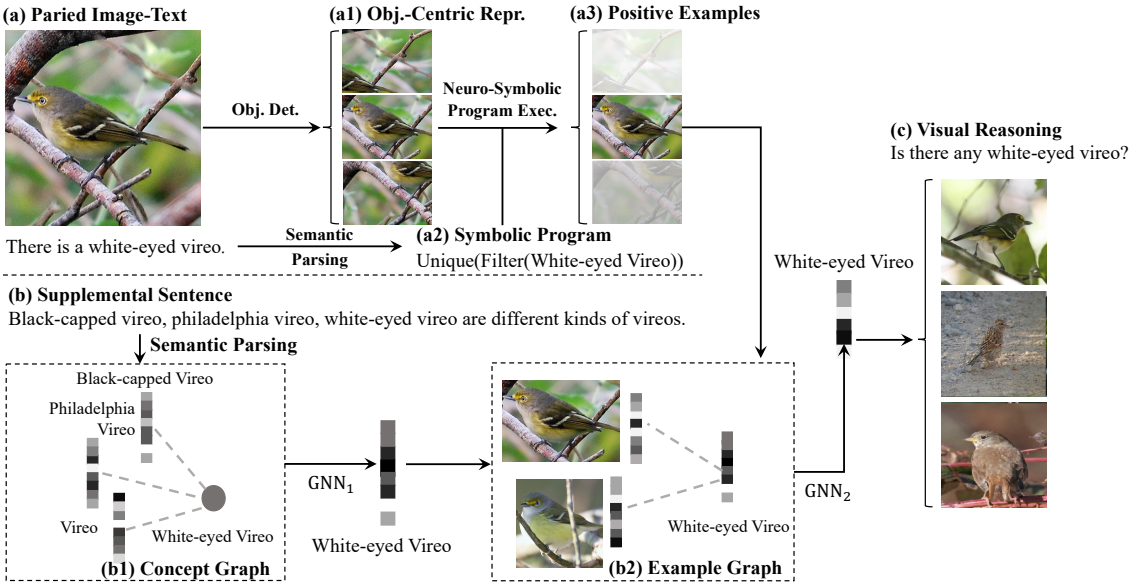


Figure 7-2: **Overview of FALCON-G.** FALCON-G starts from extracting object-centric representations and parse input sentences into semantic programs. It executes the program to locate the objects being referred to. The example objects, together with the reconstructed concept graphs are fed into two GNNs sequentially to predict the concept embedding for the "white-eyed vireo". The derived concept embedding can be used in downstream tasks such as visual reasoning.

Problem formulation. Each concept learning task is a 4-tuple (c, X_c, D_c, T_c). Denote c as the novel concept to be learned (e.g., *red*). Models learn to recognize red objects by looking at paired images x_i and sentences y_i: $X_c = \{(x_i, y_i)\}$. Optionally, supplementary sentences $D_c = \{d_i\}$ describe the concept c by relating it to other known concepts. After learning, the model will be tested on downstream tasks. In this chapter, we specifically focus on visual reasoning: the ability to answer questions about objects in the testing set T_c, which is represented as pairs of images and questions.

There are two possible options to approach this problem. One is manually specifying rules to compute the representation for the new concept. In this chapter, we focus on a meta-learning approach: to build a system that can learn to learn new concept. Our training data is data tuples for a set of training concepts (base concepts, C_{base}). After training, the system is evaluated on a collection of novel concepts (C_{test}). That is, we will provide our system with X_c and D_c for a novel concept c, and test it on visual reasoning data T_c. Thus, the system works in a continual learning fashion: the description of a new concept depends on a previously learned concept.

Overview. Fig. 7-2 gives an overview of our proposed model, FALCON. Our key idea is to represent each concept as an axis-aligned box in a high-dimensional space (the "box

embedding space", Section 7.2.1). Given example images x_i and descriptions y_i (Fig. 7-2a), FALCON interprets the referential expression in y_i as a symbolic program (Fig. 7-2a2). An neuro-symbolic reasoning module executes the inferred program to locate the object being referred to (Fig. 7-2a3), see Section 7.2.2. Meanwhile, supplementary descriptions D_c (Fig. 7-2b) will be translated into relational representations of concepts (Fig. 7-2b1), i.e., how the new concept c relates to other known concepts. Based on the examples of the novel concept and its relationships with other concepts, we formulate the task of novel concept learning as learning to infer the best concept embedding for c in the box embedding space, evaluated on downstream tasks (Fig. 7-2c). Once the model learned, we will be using the same neuro-symbolic module for answering questions in the testing set T_c.

7.2.1 Visual Representations and Embedding Spaces

Given the input image, our model extracts an object-based representation for the scene (Fig. 7-2a1). This is done by first using a pretrained Mask-RCNN [He *et al.*, 2017] to detect objects in the image. The mask for each object, paired with the original image is sent to a ResNet-34 [He *et al.*, 2015] model to extract its visual representation, as a feature vector. For each pair of objects, we will concatenate their feature vectors to form the relational representation between this pair of objects.

The feature vector of each object will be mapped into a geometric embedding space with a fully-connected layer. Each object-based concepts (such as object categories, colors) will be mapped into the *same* embedding space. Similarly, the relational representation of object pairs, as well as relational concepts (such as spatial relationships) will be mapped into another space. Here, we focus on the *box embedding space* [Vilnis *et al.*, 2018] since it naturally models the entailment relationships between concepts. It is also worth noting that other embedding spaces are also compatible with our framework, and we will empirically compare them in Section 7.3.

Box embedding. We slightly extend the box embedding model for knowledge graphs [Vilnis *et al.*, 2018] into visual domains. Throughout this section, we will be using object-based concepts as an example. The idea can be easily applied to relational concepts as well.

Each object-based concept is a tuple of two vectors: $e_c = (\text{Cen}(e_c), \text{Off}(e_c))$, denoting center and offset of the box embedding, both in $\mathbb{R}^d$, where d is the embedding dimension. Intuitively, each concept corresponds to an axis-aligned box (hypercube) in the embedding space, centered at $\text{Cen}(e_c)$ and have edge length $2 \cdot \text{Off}(e_c)$. Each object o is also a box: $(\text{Cen}(e_o), \delta)$, centered at $\text{Cen}(e_o)$ with a fixed edge length of $\delta = 10^{-6}$. We constrain that all boxes should reside within $[-\frac{1}{2}, \frac{1}{2}]^d$.

For each box embedding (either an embedding of an object, or a concept), we define helper functions $\text{Min}(e) = \text{Cen}(e) - \text{Off}(e)$ and $\text{Max}(e) = \text{Cen}(e) + \text{Off}(e)$ to denote the lower- and upper-bound of the a box e. The denotational probability is the volume of the box: $\Pr[e] = \prod_i(\text{Max}(c)_i - \text{Min}(c)_i)$, where subscript i denotes the i-th dimension of vectors. Obvsiouly, $\Pr[e] \in [0, 1]$. Moreover, we define the joint distribution of two boxes e_1 and e_2 as the volume of their intersection: $\Pr[e_1 \cap e_2] = \prod_i \max((M(e_1, e_2)_i - m(e_1, e_2)_i), 0)$, where $M(e_1, e_2) = \text{Max}(e_1) \vee \text{Max}(e_2), m(e_1, e_2) = \text{Min}(e_1) \wedge \text{Min}(e_2))$. $\wedge$ and $\vee$ are element-wise max and min operators. Finally, we define the conditional probability $\Pr[e_1|e_2] = \Pr[e_1 \cap e_2]/\Pr[e_2]$. This notation is useful in classifying objects: we will use $\Pr[e_{red}|e_{obj}]$ as the classification confidence of *obj* being *red*. Similarly, it also handles entailment of concepts: $\Pr[e_{c'}|e_c]$ readily denotes the probability that concept c entails concept c'. This well suits our need for modeling entailment relations between concepts: in Fig. 7-2, the supplemental sentence implies "*white-eyed vireo* entails *vireo*". In box embedding, this means: any instance inside the box "white-eyed vireo" should also be inside the box "vireo." We leave detailed discussions and comparisons with other embedding spaces in Appendix Section G.3.4.

The formulation above has non-continuous gradient at the edges of the box embeddings, following [Li, Xiang and Vilnis, Luke and Zhang, Dongxu and Boratko, Michael and McCallum, Andrew, 2019], we "smooth" the computation by replacing all $\max(\cdot, 0)$ operations with the softplus operator: $\text{softplus}(x) = \tau \log(1 + \exp(x/\tau))$, where τ is a scalar hyperparameter.

7.2.2 Neuro-Symbolic Reasoning

The neuro-symbolic reasoning module processes all linguistic inputs by translating them into a sequence of queries in the concept embedding space, organized by a program layout. Specifically, all natural language inputs, including the sentences y_i paired with the image examples, the supplementary sentences d_i, and questions in T_c will be parsed into symbolic programs, using the same grammar as [Han *et al.*, 2019]. Visually grounded questions including y_i and questions in T_c will be parsed into programs with hierarchical layouts. They contain primitive operations such as `Filter` for locating objects with a specific color or shape, and `Query` for query the attribute of a specified object. Appendix Section G.3 contains a formal definition of grammar. Meanwhile, supplementary sentences d_i will be translated as conjunctions of fact tuples: e.g., (red, is, color), which denotes the extra information that the new concept *red* is a kind of *color*. We use pre-trained semantic parsers to translate the natural language inputs into such symbolic representations.

Next, we locate the object being referred to in each (x_i, y_i) pair. This is done by executing the program on the extracted object representations and concept embeddings.

Specifically, we apply the neuro-symbolic program executor preseted in Chapter 6. It contains deterministic functional modules for each primitive operations. The execution result is a distribution p_j over all objects j in image x_i, interpreted as the probability that the j-th object is being referred to.

Recall that the high-level idea of the neuro-symbolic program executor is to smooth the boolean value in deterministic program execution into scores ranging from 0 to 1. For example, in Fig. 7-2, the execution result of the first `Filter[`*white-eyed vireo*`]` operation yields to a vector v, with each entry v_i denoting the probability that the i-th object is red. The scores are computed by computing the probability $\Pr[e_{white-eyed-vireo}|e_i]$ for all object embeddings e_i in the box embedding space. The output is fully differentiable w.r.t. the concept embeddings and the object embeddings. Similarly, the concept embeddings can be used to answer downstream visual reasoning questions.

7.2.3 Learning to Learn Concepts

The embedding prediction module takes the located objects $\{o_i^{(c)}\}$ of the novel concept c in the image, as well as the relationship between c and other concepts as input. It outputs a box embedding for the new concept e_c. Here, the relationship between the concepts is represented as a collection of 3-tuples (c, c', rel), where c' is an known concept and rel denotes the relationship between c and c'.

At a high level, the inference process has two steps. First, we infer N candidate box embeddings based on their relationship with other concepts, as well as the example objects. Next, we select the concept embedding candidate e_c with the highest data likelihood: $\prod_{i=1}^{M} \Pr[e_c|o_i^{(c)}]$, where i indexes over M example objects. The data likelihood is computed by the denotational probabilities. Following we discuss two candidate models for generating candidate concept embeddings.

Graph neural networks. A natural way to model relational structures between concepts and example objects is to use graph neural networks [Hamilton *et al.*, 2017]. Specifically, we represent each concept as a node in the graph, whose node embeddings are initialized as the concept embedding e_c. For each relation tuple (c, c', rel), we connect c and c'. The edge embedding is initialized based on their relationships rel. We denote the derived graph as $G_{concept}$ (Fig. 7-2 (b1)). We also handle example objects in a similar way: for each example object $o_i^{(c)}$, we initialize a node with its visual representation, and connect it with c. We denote the derived graph as $G_{example}$ (Fig. 7-2 (b2)).

To generate candidate concept embeddings, we draw N initial guesses $e_c^{i,0}, i = 1, \cdots N$ i.i.d. from the a prior distribution p_θ. For each candidate $e_c^{i,0}$, we feed it to two graph neural networks sequentially:

$$e_c^{i,0} \sim p(\theta), \qquad e_c^{i,1} = \text{GNN}_1(e_c^{i,0}, G_{concept}),$$
$$e_c^{i,2} = \text{GNN}_2(e_c^{i,1}, G_{example}), \qquad e_c = e_c^{k,2}; k = \arg\max_i \prod_{j=0}^{M} \Pr[e_c^{i,2}|o_i^{(c)}].$$

Note that GNN_1 and GNN_2 are different GNNs. We use the output e_c as the concept embedding for the novel concept c. We denote this model as FALCON-G.

Recurrent neural networks. A variant of our model, FALCON-R, replaces the graph neural network in FALCON-G with recurrent neural networks. Specifically, we process each edge in the graph in a sequential order. We first sample $e_c^{i,0}$ from $p(\theta)$ and use it to initialize the hidden state of an RNN cell RNN_1. Next, for each edge (c, c', rel) in $G_{concept}$, we concatenate the concept embedding of c' and the edge embedding for rel as the input. They are fed into RNN_1 sequentially. We use the last hidden state of the RNN as $e_c^{i,1}$. We process all edges in $G_{example}$ in the same manner, but with a different RNN cell RNN_2. Since all edges in a graph are unordered, we use random orders for all edges in both graphs. The implementation details for both the GNN and the RNN model can be found in Appendix G.1.

Training objective. We evaluate the inferred e_c on downstream tasks. Specifically, given an input question and its paired image, we execute the latent program recovered from the question based on the image, and compare it with the groundtruth answer to compute the loss. Note that since the output of the neuro-symbolic program executor is fully differentiable w.r.t. the visual representation and concept embeddings, we can use gradient descent to optimize our model. The overall training objective is comprised of two terms: one for question answering $\mathcal{L}_{QA}$ and the other for the prior distribution matching: $\mathcal{L}_{meta} = \mathcal{L}_{QA} - \lambda_{\text{prior}} \log p_\theta(e_c)$ is the final loss function.

7.2.4 Training and Testing Paradigm

The training and testing pipeline for FALCON consists of three stages, namely the pre-training stage, the meta-training stage, and the meta-testing stage. We process a dataset by splitting all concepts into three groups: C_{base}, C_{val}, and C_{test}, corresponding to the base concepts, validation concepts, and test concepts. Recall that each concept in the dataset is associated with a 4-tuple (c, X_c, D_c, T_c). In pre-training stage, our model is trained on all concepts c in C_{base} using T_c, to obtain all concept embeddings for $c \in C_{base}$. This step is the same as the concept learning stage in Chapter 6, except that we are using a geometric embedding space. In the meta-training stage, we fix the concept embedding for all $c \in C_{base}$, and train the GNN modules (in FALCON-G) or the RNN modules (in FALCON-R) with our meta-learning objective $\mathcal{L}_{meta}$. In this step, we randomly sample task tuples (c, X_c, D_c, T_c)

from concepts in C_{base}, and use concepts in C_{val} for validation and model selection. In the meta-testing stage, we evaluate models by its downstream task performance on concepts drawn from C_{test}. For more details including hyperparameters, please refer to Appendix Section G.2.

7.3 Experiments

7.3.1 Dataset

We procedurally generate meta-training and testing examples. This allows us to have fine-grained control over the dataset distribution. At a high-level, our idea is to generate synthetic descriptive sentences and questions based on the object-level annotations of images and external knowledge bases such as the bird taxonomy. We compare FALCON with end-to-end and other concept-based method on two datasets: CLEVR [Johnson *et al.*, 2017a] and CUB [Wah *et al.*, 2011].

CUB Dataset. The CUB-200-2011 dataset [CUB; Wah *et al.*, 2011] contains 12K images of 200 categories of birds. We split these 200 classes into 100 base, 50 validation and 50 test classes following Hilliard *et al.* [2018]. We expand the concept set by including their hypernyms in the bird taxonomy [Sullivan *et al.*, 2009], resulting in 211 base, 83 validation, and 72 test concepts. We consider the two concept relations in CUB: *hypernym* and *cohypernym* (concepts that share the same hypernym). Examples could be found in Fig. 7-4.

CLEVR dataset. We also evaluate our method on the CLEVR dataset [Johnson *et al.*, 2017a]. Each image contains 3D shapes placed on a tabletop with various properties – shapes, colors, types of material and sizes. These attribute concepts are grouped into 10 base, 2 validation and 3 test concepts. We create four splits of the CLEVR dataset with different concept split. We generate the referential expressions following Liu *et al.* [2019a] and question-answer pairs following Johnson *et al.* [2017a]. Examples could be found in Fig. 7-4. We consider only one relation between concepts: *hypernym*. For example, *red* is a kind of *color*.

GQA dataset. Our last dataset is thee GQA dataset [Hudson and Manning, 2019a], which contains image associated with a scene graph annotating the visual concepts in the scene. We select a subset of 105 most frequent concepts. These concepts are grouped into 89 base concepts, 6 validation and 10 test concepts. Examples generated in this dataset can be found

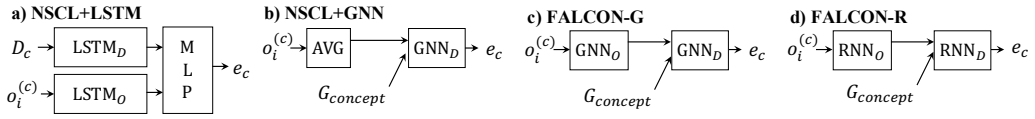


Figure 7-3: **Different models for predicting concept embedding e_c in FALCON.** They differ in how they process D_c: supplemental sentences, $G_{concept}$: concept graphs, and $o_i^{(c)}$: objects being referred to.

in Fig. 7-4. We consider only one relation between concepts: *hypernym*, e.g. *standing* is a kind of *pose*.

7.3.2 Baseline

We consider two kinds of approaches: *end-to-end* and *concept-centric*.In *end-to-end* approaches, example images and other descriptive sentences (X_c and D_c) are treated as an additional input to the system when responding to each test query in T_c. In this sense, the system is not "learning" the novel concept c as an internal representation. Instead, it reasons about images and texts that is related to c and has been seen before, in order to process the queries. Their implementation details can be found in Appendix Section G.1.

CNN+LSTM is the first and simplest baseline, where ResNet-34 [He *et al.*, 2015] and LSTM [Hochreiter and Schmidhuber, 1997] are used to extract image and text representations, respectively. They are fed into a multi-layer perceptron to predict the answer to test questions.

MAC [Hudson and Manning, 2018], is a state-of-the-art end-to-end method for visual reasoning. It predicts the answer based on dual attention over images and texts.

We also consider a group of *concept-centric* baselines, clarified in 7-3. Like FALCON, they all maintain a set of concept embeddings in their memory. The same neuro-symbolic program execution model is used to answer questions in T_c. This isolates the inference of novel concept embeddings.

NSCL+LSTM, the first baseline in this group uses the same semantic parsing and program execution module to derive the example object embeddings $o_i^{(c)}$ as FALCON. Then, it uses an LSTM to encode all $o_i^{(c)}$ into a single vector, and uses a different LSTM to encode the natural language sentences in D_c into another vector. These two vectors are used to compute the concept embedding e_c.

NSCL+GNN is similar to NSCL+LSTM except that supplemental sentences D_c are first parsed as conceptual graphs. Next, we initialize the concept embedding in the graph by taking the "average" of visual examples. We then use the same GNN module as FALCON on the conceptual graph $G_{concept}$ to derive e_c. Note that NSCL+GNN only use a single GNN corresponding to $G_{concept}$, while FALCON-G uses two GNNs–one for concept graphs and

the other for example graphs.

We test concept-centric models on two more embedding spaces besides the box embedding.

Hyperplane: object o and concept c correspond to vectors in $\mathbb{R}^d$, denoted as e_o and e_c, respectively. We define $\Pr[e_o|e_c] = \sigma(e_o^T e_c/\tau - \gamma)$, where σ is the sigmoid function, $\tau = 0.125$, and $\gamma = 2d$.

Hypercone: object o and concept c correspond to vectors in $\mathbb{R}^d$, denoted as e_o and e_c, respectively. We define $\Pr[e_o|e_c] = \sigma((\langle e_o, e_c\rangle - \gamma)/\tau)$, where $\langle\cdot,\cdot\rangle$ is the cosine similarity function, and $\tau = 0.1, \gamma = 0.2$ are scalar hyperparameters. This embedding space has been proved useful in concept learning literature [Gidaris and Komodakis, 2018].

7.3.3 Results on CUB

We evaluate model performance on downstream question-answering pairs in T_c for all test concepts $c \in C_{test}$. There is only one example image for each concept ($|X_c| = 1$). Visual reasoning questions on CUB are simple queries about the bird category in this image. Thus, it has minimal visual reasoning complexity and focuses more on the fast learning of novel concepts.

The results are summarized in Table 7.1, with qualitative examples in Fig. 7-4. In general, our model outperforms all baselines, suggesting the effectiveness of our concept embedding prediction. FALCON-R with box embeddings achieves the highest performance. We conjecture that this is because box embeddings can better capture hierarchies and partial order relationships between concepts, such as *hypernyms*. NSCL+LSTM and GNN models have inferior performance with box embeddings, probably because their neural modules cannot handle complex geometric structures like box containment. Unlike them, we use a sample-update-evaluate procedure to predict box embeddings.

By comparing concept-centric models based on the same embedding space (e.g., box embeddings), we see that translating the descriptive sentences D_c into concept graphs significantly improves the performance. Thus, NSCL+LSTM has an inferior performance than other methods. NSCL-GNN and FALCON-G are very similar to each other, except that in NSCL-GNN, the initial embedding of the concept e_c is initialized based on the object examples, while in FALCON-G, we use multiple random samples to initialize the embedding and select the best embedding based on the likelihood function. Moreover, FALCON-G and FALCON-R has similar performance, and it is worth noting that FALCON-R has the advantage that it can perform online learning. That is, edges in $G_{concept}$ and $G_{example}$ of concept c can be gradually introduced to the system to improve the accuracy of the corresponding concept embedding e_c.

We summarize our key findings as follows. First, compared with end-to-end neural base-

Model(Type)	QA Accuracy		
CNN+LSTM (E2E)	51.37		
MAC (E2E)	73.88		
	Box	Hyperplane	Hypercone
NSCL+LSTM (C.C.)	79.53	71.71	72.48
NSCL+GNN (C.C.)	78.50	67.60	72.82
FALCON-G (C.C.)	**81.33**	74.11	68.44
FALCON-R (C.C.)	79.83	72.82	69.74

Table 7.1: **Fast concept learning performance on CUB.** We use "E2E" for end-to-end methods and "C.C." for concept-centric methods. Our model FALCON-G and FALCON-R outperforms all baselines.

Model(Type)	QA Accuracy		
CNN+LSTM (E2E)	51.50		
MAC (E2E)	73.55		
	Box	Hyperplane	Hypercone
NSCL+LSTM (C.C.)	72.23	63.18	68.48
NSCL+GNN (C.C.)	73.38	62.00	72.91
FALCON-G (C.C.)	**76.37**	64.60	70.42
FALCON-R (C.C.)	75.84	63.32	69.69

Table 7.2: **Fast concept learning with only visual examples on CUB.** The task is also referred to as one-shot learning. In this setting, NSCL+GNN will fallback to a Prototypical Network [Snell *et al.*, 2017].

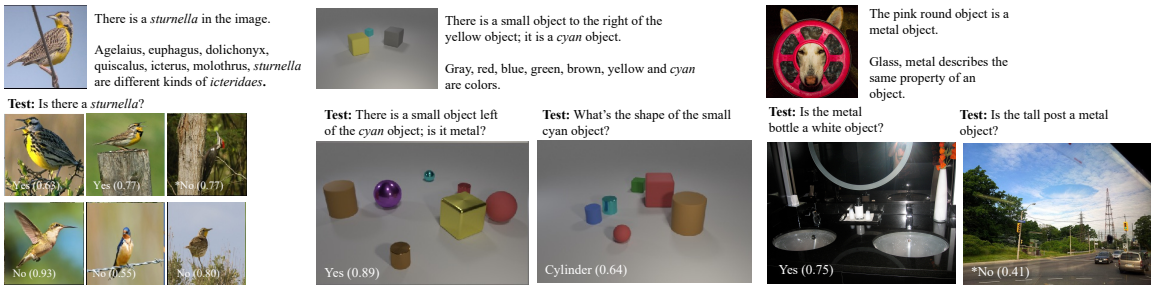


Figure 7-4: **Qualitative results of FALCON-G on CUB and CLEVR.** The bottom left corner of each images shows the model prediction and the confidence score. $*$ marks a wrong prediction.

lines (CNN+LSTM, MAC), explicit concept representation (NSCL, FALCON) is helpful (Table 7.3). Second, comparing modules for predicting concept embeddings from supplemental sentences D_c: LSTMs over texts D_c (NSCL+LSTM) v.s. GNNs over concept graphs $G_{concept}$ (NSCL+GNN, FALCON-G), we see that, explicitly representing supplemental sentences as graphs and using GNNs to encode the information yields the larger improvement. Third, comparing modules for predicting concept embeddings from visual examples $o_i^{(c)}$: heuristically taking the average (NSCL+GNN) v.s. GNNs on example graphs (FALCON-G), we see that using example graphs is better. These findings will be further supported by results on the other two datasets.

Case study: few-shot learning. Few-shot learning (i.e., learning new concepts from just a few examples) is a special case of our fast concept learning task, where $D_c = \varnothing$, i.e., there is no supplemental sentences to relate the novel concept c with other concepts.

Results for this few-shot learning setting are summarized in Table 7.2. Our model FALCON outperforms other baselines with a significant margin, showing the effectiveness of our sample-update-select procedure for inferring concept embeddings. In this setting, NSCL+GNN computes the novel concept embedding solely based on the "average" of visual examples, and thus being a Prototypical Network [Snell *et al.*, 2017]. Note that it cannot work with the box embedding space since the offset of the box cannot be inferred from just one input example.

Extension: continual learning. Another advantage of concept-centric models is that it supports the cumulative growth of the learned concept set. That is, newly learned concepts during the "meta-testing" stage can serve as supporting concepts in future concept learning. This is not feasible for end-to-end approaches as they require all concepts referred to in the supplemental sentence in the base concept set C_{base}. We extend concept-centric approaches to this continual learning setting, where test concepts C_{test} are learned incrementally, and

new concepts might relate to another concept in C_{test} that has been learned before. In this setup, FALCON-G achieves the highest accuracy (79.67%), outperforming the strongest baseline NSCL+LSTM by 4.46%. Details are in Appendix Section G.3.1.

Ablation: the number of base concepts. We examine how the number of base concepts contributes to the model performance. We design a new split of the CUB dataset containing 50 training species (130 base concepts), compared to 100 training species (211 base concepts) of the original split. Our FALCON-G model's accuracy drops from 81.33% to 76.32% when given fewer base concepts. This demonstrates that transfering knowledge from concepts already learned is helpful. See Appendix G.4.1.

Ablation: the number of related concepts in supplemental sentences. We provide an ablation on the effects of the number of related concepts in supplemental sentences. We use supplemental sentences containing 0%(no supp. sentence), 25%, 50%, 75%, and 100%(the original setting) of all related concepts. Our model FALCON-G yields an accuracy of 76.37%, 80.20%, 80.78%, 81.20%, 81.33%, respectively. This result shows that more related concepts included in supplemental sentences lead to higher accuracy. Details are in Appendix G.4.2.

7.3.4 Results on CLEVR

We use a similar data generation protocol for the CLEVR dataset as CUB. While CLEVR contains concepts with simpler appearance variants, it has a more complicated linguistic interface. More importantly, all concepts in CLEVR are disentangled and thus allow us to produce compositional object appearances. We will exploit this feature to conduct a systematical study on concept learning from biased data. Our framework FALCON provides a principled solution to learning from biased data, which has been previously studied in Geirhos *et al.* [2019] and Han *et al.* [2019].

Our main results are summarized in Table 7.3. Fig. 7-4 provides qualitative examples. All concept-centric models use the box embedding space. We also compare model performance on the "few-shot" learning setting, where no supplemental sentences are provided. Due to the compositional nature of CLEVR objects, learning novel concepts from just a single example is hard, and almost always ambiguous: the model can choose to classify the examples based on either its shape, or color, or material. As a result, all models have a relatively poor performance when there are no supplementary sentences specifying the kind of novel concept (color/shape/etc.).

Case study: learning from biased data. We provide a systematical analysis on concept learning from biased data. In this setup, each test concept c is associated with another test

Model	Example Only	Example+Supp.	Δ
CNN+LSTM	41.92	42.69	0.77
MAC	61.81	62.05	0.24
NSCL+LSTM	56.82	61.35	4.53
NSCL+GNN	65.01	84.11	19.11
FALCON-G	68.47	**88.40**	19.93
FALCON-R	68.57	87.36	18.78

Table 7.3: **Fast concept learning performance on CLEVR.** We compare both Example only and the Example +Supp. settings. The results are evaluated and averaged on the four splits of the dataset. See the main text for analysis.

concept c' (both unseen during training). We impose the constraint that all example objects of concept c in X_c also has concept c'. Thus, c and c' is theoretically ambiguous given only image examples. The ambiguity can be resolved by relating the concept c with other known concepts. Overall, our model, FALCON-G achieves the highest accuracy (88.29%) in this setting (MAC: 61.93%, NSCL+GNN 84.83%). Moreover, our model shows the most significant performance improvement after reading supplemental sentences. Details can be found in Appendix Section G.3.2.

7.3.5 Results on GQA

Following a similar data generation protocol as the previous datasets, we generate questions on the GQA dataset that ask whether a linguistically grounded object in the picture is associated with a novel concept. Since each GQA picture contains a large number of concepts in different image regions, it provides a more complex linguistic interface than the CUB dataset and more diverse visual appearances than the CLEVR dataset. We exploit this advantage to test our model's performance on real-world images and questions in the GQA dataset.

Our main results are shown in Table G.9 in the Appendix Section G.3.3. Fig. 7-4 provides qualitative examples. Overall, our model achieves the highest accuracy (55.96%) in the setting with supplementary sentences (MAC: 54.99%, NSCL+LSTM: 55.41%). Due to the visual complexity in the GQA dataset, the exact decision boundary of a concept is hard to determine. Details are in Appendix Section G.3.3.

7.4 Related Work

Visual concept learning and visual reasoning. Visual reasoning aims to reason about object properties and their relationships in given images, usually evaluated as the question-

answering accuracy [Johnson *et al.*, 2017a; Hudson and Manning, 2018; Mascharka *et al.*, 2018; Hu *et al.*, 2018]. Recently, there has been an increasing amount of work has been focusing on using neuro-symbolic frameworks to bridge visual concept learning and visual reasoning [Yi *et al.*, 2018; Mao *et al.*, 2019; Li *et al.*, 2020b]. The high-level idea is to disentangle concept learning: association of linguistic units with visual representations, and reasoning: the ability to count objects or make queries. Han *et al.* [2019] recently shown how jointly learning concepts and metaconcepts can help each other. Our work is an novel approach towards making use of the metaconcepts in a meta-learning setting aiming at boost the learning of novel concepts based on known ones.

Few-shot visual learning. Recent work has studied learning to classify visual scene with very limited labeled examples [Vinyals *et al.*, 2016; Sung *et al.*, 2018; Snell *et al.*, 2017] or even without any example [Wang *et al.*, 2018c; Kampffmeyer *et al.*, 2019; Tian *et al.*, 2020]. For few-shot learning, existing work proposes to compare the similarity, such as cosine similarity [Vinyals *et al.*, 2016] and Euclidean distance [Snell *et al.*, 2017], between examples, while [Sung *et al.*, 2018] introduces a learnable module to predict such similarities. In addition, [Gidaris and Komodakis, 2018] learns a weight generator to predict the classifier for classes with very few examples. Ravi and Larochelle [2017]; Finn *et al.* [2017]; Nichol *et al.* [2018] address this problem by learning the initialization for gradient-based optimization. Santoro *et al.* [2016] used external memory to facilitate learning process, while [Munkhdalai and Yu, 2017] uses meta-knowledge among task for rapid adaptation. Our module design is inspired by these work, but we use a language interface: novel concepts are learnt from paired images and texts and evaluated on visual reasoning tasks.

Geometric embeddings. In contrast to representing concepts in vector spaces [Kiros *et al.*, 2014], a geometric embedding framework associates each concept with a geometric entity such as a Gaussian distribution [Vilnis and McCallum, 2015], the intersection of hyperplanes [Vendrov *et al.*, 2016; Vilnis *et al.*, 2018], and a hyperbolic cone [Ganea *et al.*, 2018]. Among them, box embeddings [Vilnis *et al.*, 2018] which map each concept to a hyper-box in the high-dimensional space, have been popular for concept representation: [Li, Xiang and Vilnis, Luke and Zhang, Dongxu and Boratko, Michael and McCallum, Andrew, 2019] proposed a smooth training objective, and [Ren *et al.*, 2020] uses box embeddings for reasoning over knowledge graphs. In this chapter, we extend the box embedding from knowledge graphs to visual domains, and compare it with other concept embeddings.

7.5 Discussion

We have presented FALCON, a meta-learning framework for fast learning of visual concepts. Our model learns from multiple naturally occurring data streams: simultaneously looking at images, reading sentences that describe the image, and interpreting supplemental sentences about relationships between concepts. Our high-level idea is to incorporate neuro-symbolic learning frameworks to interpret natural language and associate visual representations with latent concept embeddings. We improve the learning efficiency by meta-learning concept embedding inference modules that integrates visual and textual information. The acquired concepts directly transfer to other downstream tasks, evaluated by the visual question answering accuracy. Overall, this chapter illustrates a meta-learning approach to acquiring neuro-symbolic concepts from multimodal observations, including images and textual descriptions.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 8

Concept Learning Across Domains and Modalities

This chapter extends the NS-CL framework introduced in Chapter 6 to learning to reason across domains and modalities, such as 3D point clouds and motion capture trajectories. The key idea is to leverage large language models (LLMs) as pretrained, general-purpose models to translate natural language queries into executable programs in a shared first-order logic language. This chapter introduces a unified framework that learn to ground and reason with concepts across domains using a differentiable, domain-independent, first-order logic–based program executor. Conceptually, the executor reformulates the ad hoc domain-specific languages introduced in Chapter 6 using first-order logic. From an implementation perspective, it extends the neural logic machines from Chapter 2 to logic programs. It demonstrates strong reasoning ability across a wide variety of tasks, including complex ones not seen during training, and can be readily applied to new domains.

The material of this chapter has been previously published in Hsu *et al.* [2023a]. Joy Hsu, in particular, contributed significantly to the materials presented in this chapter.

8.1 Introduction

The power of language lies in its generalization based on abstraction. A single concept, such as *left*, can be used across domains: in perception, we recognize the *left* leg from a chair's 3D point cloud; in navigation, we go home by turning *left* around the cafe at the end of the block; in manipulation, we pick and place a mug on the *left* of a plate. *Left* has domain-specific groundings in each of these domains, but the concept symbol itself as an abstraction serves as a basis for complex and multi-step reasoning across domains.

Machine systems for visual reasoning also benefit from the disentanglement of concept

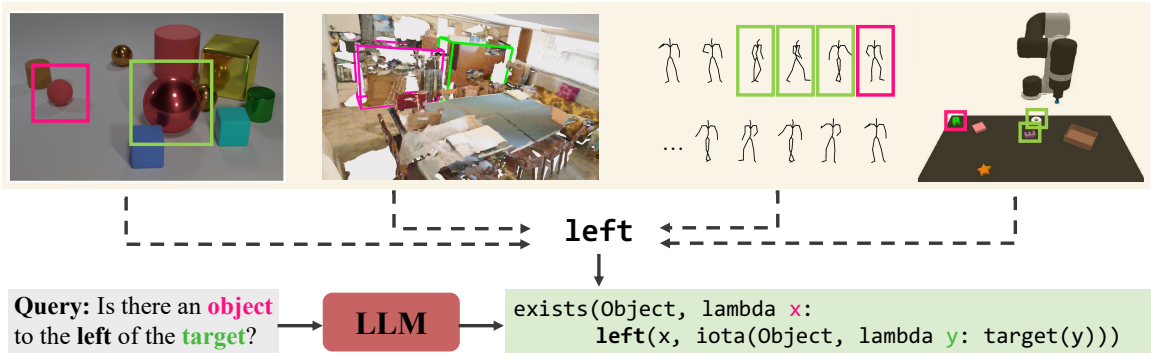


Figure 8-1: **Overview of LEFT.** LEFT is a unified concept learning and reasoning framework that grounds modular concepts across domains, and flexibly reasons with concepts across tasks with a foundation model.

learning and reasoning. Most recently, models such as VisProg [Gupta and Kembhavi, 2023] and ViperGPT [Surís *et al.*, 2023] have leveraged large language models (LLMs) to produce programs based on language queries. For example, to answer "what's the color of the cat?", the program will first locate the cat, then query its color. Such programs are executed with pre-defined and pre-trained vision-language models (e.g., open-vocabulary segmentation models) for grounding and question-answering. These works perform well on image-based reasoning, though surprisingly, despite the generalization seemingly offered for free by language, less success has been shown in other domains.

What's *left* to make visual reasoning systems work across domains? We identify two key shortcomings of existing methods. First, they are *inference-only*, building exclusively upon pre-trained models to interpret concepts such as *left* during program execution. Therefore, in domains where data is scarce (e.g., 3D, human motion, or robot action), pretrained models, if there exist any, do not perform as well. Thus, we need trainable systems that learn the domain-specific groundings of concepts to execute across domains. Unfortunately, these visual reasoning models also cannot be made trainable due to their second shortcoming: they rely on a non-differentiable program executor to compose these grounding modules. Therefore, though *left* may have different groundings in 3D scenes and robot actions compared to 2D images, these modules will not be able to adapt through additional training in those domains.

We address these issues by proposing Logic-Enhanced Foundation Models (LEFT), a model that *learns* to ground and reason with concepts across domains. LEFT also leverages LLMs for language parsing; but unlike prior work, LEFT has trainable concept grounding modules that learn from domain data, made possible via a differentiable, domain-independent, first-order logic–based program executor. Inspired by previous works that combine symbolic programs and deep models via differentiable execution (Chapter 6),

our design no longer requires manually designed domain-specific languages and facilitates generalization across domains. Specifically, LEFT's LLM interpreter takes as input language queries, resolves textual ambiguities, and outputs a non-ambiguous program represented in a general reasoning language of first-order logic, which is shared across domains and tasks. LEFT's executor then executes the logic program with learnable domain-specific grounding modules, which are automatically initialized with concepts generated by the LLM.

LEFT enjoys the advantages of strong performance and data efficiency across domains due to its modular structure. Within a single domain, LEFT generalizes zero-shot to unseen and complex tasks, by leveraging general reasoning capabilities from the LLM interpreter, and effectively recomposing learned, grounded concepts with the generic first-order logic executor. LEFT can be viewed as a generalized framework of VisProg and ViperGPT; in domains where pre-trained models are available and training is not required (e.g., 2D images), LEFT can similarly be used inference-only.

We validate LEFT's performance on four different domains and seven tasks, ranging from 2D question answering, 3D referring expression comprehension, temporal sequence reasoning, and robotic manipulation. A general LEFT model significantly outperforms prior task-specific monolithic methods in performance and in data efficiency settings, and yields comparable performance to prior neuro-symbolic methods while requiring no pre-defined program implementations for each domain. Importantly, the unified LEFT framework can perform concept learning across domains, as well as show zero-shot transfer to three challenging and unseen reasoning tasks. In contrast, inference only LLM-based methods and general vision-language models fail to generalize.

8.2 Logic-Enhanced Foundation Model (LEFT)

Our Logic-Enhanced Foundation Model (LEFT) is a unified framework for concept learning and reasoning across different domains and tasks. It integrates large language models with differentiable logic modules, and modular neural networks for grounding concepts in each modality. Shown in Fig. 8-2, our system consists of three main components:

1. The first is a domain-independent LLM language interpreter, which generates first-order logic queries to the FOL execution engine (Section 8.2.1). The generated symbolic programs are represented with a hierarchical first-order logic structure.

2. The second is a domain-independent FOL executor, which executes the logic programs differentiably based on features of entities in the domain: objects, relations, actions, etc. (Section 8.2.2). LEFT's executor is fully differentiable, which allows for backpropagation.

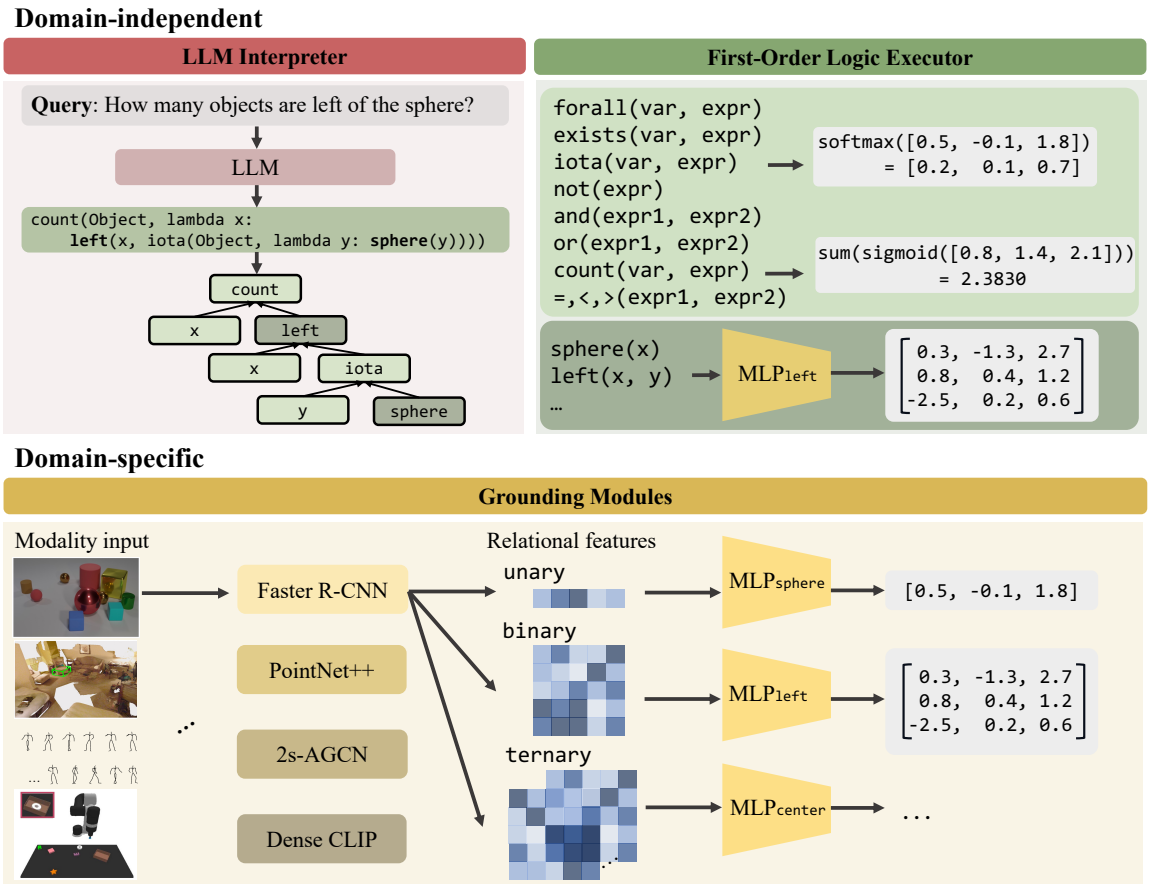


Figure 8-2: **Domain-independent and domain-specific modules in LEFT.** At the core of LEFT is a foundation model-powered language interpreter that generates first-order logic (FOL) programs from language queries, a domain-independent, differentiable FOL executor that operates on all types of entities, and domain-specific grounding modules that connect symbols in the first-order logic language with domain-specific features.

3. The third is the domain-specific grounding modules, which consist of encoders for each modality that extract entity-centric and relational features, as well as the corresponding concept embeddings implemented as modular neural networks (Section 8.2.3). The concept grounding modules are initialized with concepts generated by the LLM interpreter.

LEFT is a framework of domain-independent reasoning modules (composed of the LLM interpreter and the FOL executor) and domain-specific grounding modules. With these components, LEFT conducts concept learning and reasoning across domains and tasks.

8.2.1 Domain-independent LLM interpreter

We leverage a pre-trained large language model as our domain-independent language interpreter. It takes language queries as inputs and generates first-order logic queries to downstream modules. The examples we used to prompt the LLMs is a minimal set of general, domain-independent examples and system-level description of syntax rules in our FOL language. The LLM handles both natural language understanding and commonsense reasoning. For example, it needs to resolve coreferences and ambiguous modifier attachments: "Look at the book on the shelf, *the blue one*, select the object left of *it*." Furthermore, it performs commonsense reasoning, such as "Find an object that can fit into a row of blue spheres." The query it generates for the downstream FOL reasoning module will be non-ambiguous programs: $\iota x.blue(x) \wedge sphere(x)$. This strategy maximally harnesses the reasoning capability of LLMs across tasks, while retaining strong generalization. We show in experiments that the LEFT interpreter can reason about analogies, resolve coreferences, and tackle complex puzzles.

Formally, given an input task query Q in language, our LLM interpreter outputs programs P, which are programs written in first-order logic that will be passed to downstream FOL executors. Each program is composed of a function name and arguments, and can be chained together hierarchically such that the output of one function can be the input argument to another function. The LLM-interpreted program contains domain-independent FOL functions, which can take outputs from domain-specific functions as arguments. The FOL functions are a general set of pre-defined programs with built-in implementations and are listed in full in Table 8.1. These functions are either logic and numeric operations (such as counting, forall, exists), or functions that handle inputs and outputs (e.g., return a text description of object categories, or execute an action output by other modules). We present the syntax and built-in components of our FOL language in the next section.

LEFT leverages GPT-3.5 [Brown *et al.*, 2020] as its LLM backbone. Importantly, the LEFT prompts are domain-independent, including simple syntax rules and examples written with general concepts such as *cat* and *apple* that do not occur in the any of our evaluation

Name	Syntax	Description
variables	$x, y, z, \cdots$	Variables that refer to entities in the domain.
$\exists, \forall$	$exists(var, expr)$	*expr* is an expression that contains the variable *var*; return true if there is at least one assignment of *var* that satisfies *expr*.
iota (ι)	$iota(var, expr)$	*expr* is an expression that contains the variable *var*; return an assignment to *var*'s that satisfies *expr*.
not	$not\ expr$	Compute the negation of an expression.
and, or	$expr_1\ and\ expr_2$	Compute the conjunction/disjunction of an expression.
count	$count(var, expr)$	Count the number of assignments to *var* that will make *expr* true.
$=, <, >$	$eq(expr_1, expr_2)$	Built-in number comparison functions.
view	$view(expr)$	*expr* is an object, e.g., $view(\iota x.blue(x))$; it changes the view direction to face the object.
describe	$describe(expr)$	*expr* is a text description, e.g., $describe(\iota c.color(c, \iota x.sphere(x)))$; it returns the description (the color of the sphere).
do	$do(expr)$	*expr* is an action, e.g., $do(\iota a.pick\text{-}up(a, \iota x.sphere(x)))$; it executes the action (pick up the sphere).
concepts	$left(x, y)$	A function that takes objects, actions, and texts as inputs and returns Boolean values.

Table 8.1: **LEFT's first-order logic language**, a general reasoning language across domains.

domains. Examples in our prompt also have minimal complexity, such as first-order logic sentences for "*Is there an apple next to the cake*", paired with short descriptions of FOL usage, such as "*to classify whether two objects have a certain property, use 'on(x, y)'.*" Additionally, we use a step-by-step prompting paradigm, which asks the LLM to first reason about the query and simplify the text in language form, then generate FOL programs.

8.2.2 Domain-independent first-order logic executor

The LEFT domain-independent executor operates on first-order logic, which serves as a general reasoning language shared across different tasks. Given the parsed FOL programs P from the LLM interpreter, the executor operates P with grounding modules to return the final answer. The programs are executed recursively to model the hierarchical structure of the FOL reasoning process. The LEFT programs are implemented as differentiable first-order logic modules in Python code. Conceptually, it reformulates the ad hoc domain-specific languages introduced in Chapter 6 using first-order logic. From an implementation perspective, it extends the neural logic machines from Chapter 2 to logic programs.

Our FOL language contains three parts: first, basic first-order logic operations such as Boolean operations and logic quantifiers; second, built-in functions such as *describe*, which generate a natural language response for an entity or a pair of entities, and *do*, which execute a robotic action; and third, domain-specific concept names, including object properties

(e.g., *cat*), object relations (e.g., *left*), and actions (e.g., *pack*). These concept predicates are automatically produced by the LLM based on the language query, instead of being selected from a pre-defined set of functions; hence, the expressiveness of LEFT is not limited by the given set of primitives such as exists and count. It can learn to execute any new functions proposed by the LLM.

For a given universe of entities, the LEFT executor executes all expressions using a tensor-based representation of truth values. For example, given a scene with N objects, the execution result of the expression $sphere(x)$, where x is an unquantified variable, can be represented as a vector of length N, where each entry i denote whether object i is a sphere. Relation expressions such as $left(x, y)$ can be represented as matrices of size $N \times N$. Whether an object has been selected can be represented similarly, as a vector of size N, where each entry i is the probability of object i being selected. In practice, values such as $sphere(x)$ and $left(x, y)$ will be predicted by domain-specific grounding neural networks. In Table 8.1, we detail the execution strategy for all operations. Essential to our goal of a general neuro-symbolic framework, LEFT only requires these listed FOL functions to reason across numerous domains, including 2D, 3D, temporal, and robotic manipulation.

The programs generated from our LLM interpreter include FOL programs along with domain-specific concepts, which are executed by our grounding modules. The LEFT executor implements these programs based on the arity (e.g., number of arguments in a function) of the first-order logic program. Unary-level functions are implemented with entity-centric features of size N for each entity from the input, and binary-level functions are implemented with relational features of size $N \times N$, etc. Notably, each domain-specific concept is initialized by the LLM, and the function implementation is executed with modular neural networks, which does not require manual definition in code.

Execution. We describe implementations for the FOL programs below. See Appendix for details.

exists*(var, expr)*: We first recursively execute *expr* and get a tensor where *var* is one of the dimensions of the tensor. We perform a max pooling over that dimension. For example, the execution of $sphere(x)$ yields a vector of length N, for $exists(x, sphere(x))$, we take the max value of the returned tensor.

iota*(var, expr)*: We recursively execute *expr* and get a vector where *var* corresponds to the dimensions of the vector. We then perform a softmax operation over the vector.

and*(*$expr_1$*, *$expr_2$*)*: The recursive execution of $expr_1$ and $expr_2$ will yield two tensors. The conjunction of two resulting tensors is done by taking the element-wise minimum operation for two tensors. The same execution strategy is used for or operations (by taking the element-wise maximum) and logical negation (by taking the element-wise negation).

count*(var, expr)*: We recursively execute *expr*, then perform a sigmoid operation followed by a sum over the vector to compute the expected number of objects.
eq*($expr_1$, $expr_2$)*: The recursive execution of $expr_1$ and $expr_2$ will yield two scalar values s_1 and s_2. We define the probability of $s_1 = s_2$ as $\sigma(\alpha \cdot (\gamma - |s_1 - s_2|))$, where $\alpha = 8$ and $\gamma = 0.25$ are fixed hyperparameters. Less than and greater than operations are implemented similarly.
left*(x, y)*: As an example of a domain-specific concept, the *left* function takes two variables as its input, and generates a score matrix of size $N \times N$. This is done by applying MLP_{left} on the binary embedding extracted by the feature extractor. Note that throughout the execution, we keep the "logit" of the predicted values instead of normalizing them to $[0, 1]$ with a sigmoidal activation. This improves the numerical stability of the execution.

8.2.3 Domain-specific grounding modules

The final component of LEFT is the domain-specific grounding modules that connect concepts in the first-order logic language with modality-specific representations. LEFT does not depend on a pre-defined set of lexical concepts, and relies on LLMs to automatically extract concept names from language in the data. Our framework then initializes domain-specific grounding modules accordingly to learn the grounding of these concepts from training data. In other words, the grounding modules do not require manual definitions, can support execution of any given query, and are realized with a generic mechanism described below. The grounding modules are implemented with modular neural networks, consisting of modality-specific encoders and concept embeddings. We first extract entity-centric and relational features with modality-specific encoder ε. For 2D images, ε is Faster R-CNN [Ren *et al.*, 2015] for encoding object-centric features and DenseCLIP [Rao *et al.*, 2022] for generating object attention masks. For 3D point clouds, ε is PointNet++ [Qi *et al.*, 2017]; and for motion sequences, ε is the Two-Stream Adaptive Graph Convolutional Network [Shi *et al.*, 2019b]. Unary features are represented as a set of features for each entity, binary features as a 2D matrix of relations between each pair of entities, and ternary relations as a 3D matrix, etc.

Given entity-centric representations, we then execute concept functions such as *sphere* and *left*(x, y). In general, we can have arbitrary neural networks that take the relational representation of entities and outputs a (soft) Boolean value from $[0, 1]$. In this chapter, we initialize a multi-layer perceptron (MLP) for each concept that has been mentioned by the large language model, to represent the concept embedding. For example, in our 2D image domain, each object i has a vector embedding obj_i extracted by a Faster R-CNN encoder. We apply MLP_{sphere} on each object representation obj_i and generate a vector of length N, which is the execution result for the expression $sphere(x)$.

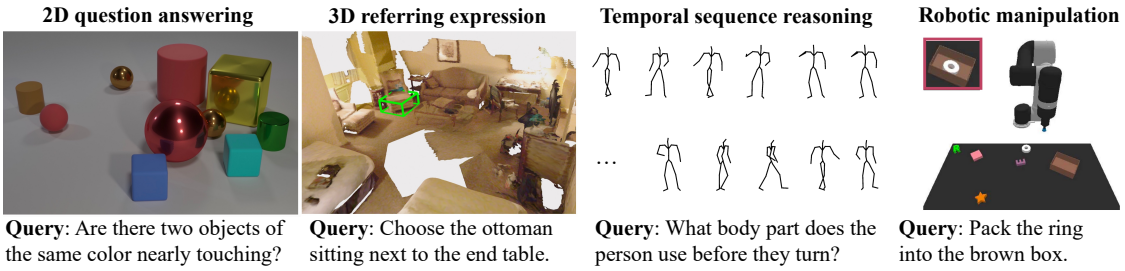


Figure 8-3: **Evaluation tasks of LEFT.** LEFT is evaluated on four broad categories of tasks, ranging from concept learning in 2D images, 3D point clouds, temporal motion sequences, and robotic manipulation.

The grounding modules are automatically initialized based on the concept name as well as *arity* proposed by the LLM. For example, if a concept is unary (e.g., *red*(x)), the grounding module will be a MLP layer that maps object representations to the aforementioned soft Boolean value scalar. For the concept *beside*, the LLM specifies that the concept takes in features of two entities, and hence LEFT creates an MLP that operates on binary features. The domain-specific feature extractors and concept MLPs are trained through backpropagation, as our executor is fully differentiable.

8.3 Experiments

We evaluate LEFT on four domains and seven tasks, and show its effectiveness in multiple settings. In Section 8.3.1, we compare LEFT to prior state-of-the-art neuro-symbolic and monolithic end-to-end methods on concept learning across four domains. Our model excels at task performance and also data efficiency, without requiring any domain-specific knowledge. Next, in Section 8.3.2, we present LEFT's zero-shot generalization performance to three novel and challenging visual reasoning tasks, and compare its reasoning capabilities to that of LLM-based decomposition frameworks and a general vision-language model. Implementation and training details for LEFT can be found in the Appendix.

8.3.1 Concept learning across domains

We train and validate LEFT on six different datasets, grouped into four domains (Figure 8-3).

2D visual concept learning. In the first domain, we focus on the task of visual question answering in 2D images: given an image and a language query (e.g., *"How many small cylinders are partially hidden?"*), the goal is to predict an answer from a set vocabulary. The

accuracy is calculated as the exact match to ground truth answers. We validate LEFT on the CLEVR and CLEVR-Humans dataset [Johnson *et al.*, 2017a], which contain images with objects rendered of different shape, size, material, and color. CLEVR contains synthetically generated queries, while the CLEVR-Humans is a human-annotated dataset with diverse natural language questions.

3D visual concept learning. The second domain evaluates 3D referring expression comprehension. The input is a set of object point clouds in the scene, and a query (e.g. *"The monitor that you would class as in the middle of the other two"*). Models are evaluated by the accuracy of selecting the correct target object. In the 3D domain where we need reference frames to disambiguate relations like *left*, the grounding of such concepts will consider the viewpoint specified in the language query, through a generic high-arity function that does not require a specific implementation. We evaluate LEFT on the ReferIt3D dataset [Achlioptas *et al.*, 2020], with 3D point clouds from ScanNet [Dai *et al.*, 2017], a collection of indoor scenes. ReferIt3D consists of two subsets, SR3D, which focuses on spatial reference with synthetically generated queries, and NR3D, with human-annotated queries. We use the NR3D subset from NS3D, which contain the same set of concepts as in SR3D.

Motion capture concept learning. The third domain contains human motion capture. In particular, we focus on the task of question answering in temporal motion sequences. Given a sequence of human motion, parameterized by joints, and a question (e.g. *"What action does the person do before they move backward and after they kick?"*), the goal is to predict the exact match answer. We train and evaluate LEFT on the HumanMotionQA dataset [Endo *et al.*, 2023], which asks questions about temporal relations and a variety of attributes, including action, direction, and body parts.

Robotic manipulation. Finally, we demonstrate that LEFT can learn action concepts that correspond to robotic motions, in the context of robotic manipulation. The input to this task is an image representing a tabletop environment, and a language instruction (e.g., *"Pack the yoshi figure into the brown box."*) The goal is to generate a sequence of actions that maximizes the average total reward over this robotic manipulation task. We validate LEFT on the CLIPort dataset [Shridhar *et al.*, 2022a], focusing on the task of sequential packing Google Scanned Objects [Downs *et al.*, 2022].

Accuracy. We compare LEFT against two categories of top-performing models across all six datasets. The first category contains neuro-symbolic models, which require a pre-defined domain-specific language and implementations for each program. The second category

	Pre-def.	CLEVR	SR3D	HumanMotionQA	Cliport
Prior NS	Yes	0.992 (NSCL)	0.627 (NS3D)	0.578 (NSPose)	0.797 (ProgramPort)
End-to-End	No	0.989 (MAC)	**0.670** (BUTD *)	0.430 (MotionCLIP)	0.708 (Cliport)
LEFT	No	**0.996**	0.620	**0.582**	**0.793**

Table 8.2: **Results comparing LEFT to prior top-performing neuro-symbolic and end-to-end works.** Our unified LEFT framework yields strong performance on all settings without any pre-defined programs.

	Pre-def.	CLEVR-Humans	NR3D
Prior NS	Yes	0.678 (NSVQA)	0.526 (NS3D)
End-to-End	No	**0.817** (MDETR)	0.430 (MVT)
LEFT	No	0.788 / 0.698	**0.496** / **0.485**

Table 8.3: **Results on natural language tasks.**

contains monolithic end-to-end methods which usually require a significant amount of data to train. Notably, our unified LEFT framework is able perform on all domains, instead of requiring specific reasoning modules for each setting.

Table 8.2 summarizes the results of different models on four datasets that involve synthetically generated language, and hence ground truth programs exist for neuro-symbolic methods and our approach. We see that across CLEVR, SR3D, HumanMotionQA, and CLIPort, LEFT achieves comparable performance to prior neuro-symbolic works [Mao *et al.*, 2019; Hsu *et al.*, 2023b; Endo *et al.*, 2023; Wang *et al.*, 2023b], and outperforms end-to-end methods in CLEVR, HumanMotionQA, and CLIPort [Hudson and Manning, 2018; Jain *et al.*, 2022; Shridhar *et al.*, 2022a]. Importantly, our LEFT model applies a unified language interpretation and program execution framework for across all domains and tasks, while prior work on neuro-symbolic learning requires domain-specific languages and reasoning modules for each dataset. This suggests that first-order logic language can serve as an effective and general language for representing language queries.

In Table 8.3, we examine LEFT on tasks with human-annotated natural language queries. The first number for LEFT represents accuracy out of all queries where the program generated by LLM is executable, while the latter number represents accuracy out of all queries, considering the percentage of LLM outputs that are not executable. We note that MDETR [Kamath *et al.*, 2021] outperforms LEFT on CLEVR-Humans, possibly because it can better capture the similarity between similar concepts such as *front* and *in-front-of*. In contrast, the representations for different concepts in LEFT are independent. The

*BUTD operates on a constrained setting, assuming 485 classes instead of the full 607 classes in ReferIt3D.

	Pre-def.	0.5%	1.5%	2.5%	5%	10%	100%
NS3D	Yes	0.426	0.520	0.556	0.591	0.600	0.627
BUTD-DETR	No	0.083	0.158	0.259	0.395	0.528	**0.670**
MVT	No	0.161	0.275	0.322	0.380	0.491	0.645
SAT	No	0.172	0.260	0.298	0.330	0.362	0.579
LEFT	No	**0.410**	**0.521**	**0.565**	**0.579**	**0.591**	0.620

Table 8.4: **Results on SR3D in data efficient settings.** LEFT performs comparably to prior neuro-symbolic works, and outperforms end-to-end works significantly.

prior neuro-symbolic method we compared against outperforms LEFT on NR3D, as it also leverages an LLM for language interpretation, but uses a set of domain-specific examples in prompts that are hand-crafted.

Data efficiency. We evaluate LEFT on data efficiency settings for CLEVR [Johnson *et al.*, 2017a] and SR3D [Achlioptas *et al.*, 2020]. LEFT significantly outperforms all end-to-end methods at every data efficient percentage point tested, and performs comparably to best-performing neuro-symbolic methods [Hsu *et al.*, 2023b; Jain *et al.*, 2022; Huang *et al.*, 2022a; Yang *et al.*, 2021; Mao *et al.*, 2019; Mascharka *et al.*, 2018; Hudson and Manning, 2018] (See Table 8.4 and Table 8.5). The percentages indicate the percentage of train data used for training compared to the full dataset.

Focusing on CLEVR as an ablation study, we note that NSCL outperforms LEFT on the 10% setting. We attribute this gap to the additional domain-specific inductive biases that NSCL leverages. In particular, NSCL assumes the existence of four attribute spaces (color, shape, material, and size), and each concept lies in exactly one attribute space. Furthermore, all concepts within the same attribute space are mutually exclusive. Therefore, the inductive biases encoded by NSCL can be seen as a combination of a hierarchical program-based reasoning mechanism, and additional domain-specific priors on concepts. By contrast, LEFT only leverages the program-based reasoning mechanism, which is general across many domains and tasks. In comparison of our model with other baselines and NSCL, we show that this general program-based reasoning mechanism is powerful enough to explain most of the data efficiency improvements.

8.3.2 Reasoning generalization across tasks

Finally, we present LEFT's ability to zero-shot generalize to unseen tasks.

CLEVR reasoning tasks. We generate three zero-shot transfer tasks based on the CLEVR datasets [Johnson *et al.*, 2017a], with 100 evaluation examples each. See Figure 8-4 for

	Pre-def.	10%	25%	50%	100%
NSCL	Yes	0.989	0.992	0.994	0.992
TbD-Net	Yes	0.541	0.560	0.912	0.991
MAC	No	0.673	0.940	0.974	0.989
LEFT	No	**0.941**	**0.991**	**0.992**	**0.996**

Table 8.5: **Results on CLEVR in data efficient settings.**

an example of each task. The CLEVR-Ref task consists of a CLEVR image and language queries, requiring LEFT to locate the exact target object. The CLEVR-Puzzle task consists of images paired with visual puzzle descriptions, which necessitates a variable assignment puzzle solving. The CLEVR-RPM task is based on Raven's Progressive Matrices, which consists of an image and a 3x3 grid describe by language. This task requires the LLM interpreter to reason about patterns in the grid by row and column to determine the attributes of the target object, and generate queries to the FOL executor to check if such an object exists in the scene. We include dataset details in the Appendix.

Accuracy. We train LEFT on the CLEVR question answering task, and report zero-shot transfer performance on the three unseen reasoning tasks. In Table 8.6, we see that LEFT achieves strong performance on all datasets. We compare LEFT performance with ground truth programs (GT programs) and with FOL programs from our LLM interpreter (LLM programs). This ablation demonstrates that there are only small drops in performance with LLM programs as compared to ground truth programs across all three complex reasoning tasks. LEFT can zero-shot transfer to novel visual reasoning tasks with LLM generated first-order logic, and effectively reuse concept embeddings, enabling flexible generalization. Importantly, we see that LEFT performs well on CLEVR-RPM, a challenging task which contains complex Raven's Progressive Matrices [Barrett *et al.*, 2018] descriptions that require the LLM interpreter to reason about patterns described by language. While the prompts given to the LLM are simple, our interpreter can solve complex tasks.

As baselines, we compare LEFT with LLM-based decomposition frameworks and general vision-language models. VisProg [Gupta and Kembhavi, 2023] and ViperGPT [Surís *et al.*, 2023] are inference only, integrated LLM with APIs frameworks, that has shown success on a variety of visual reasoning tasks. In Table 8.6, we see that LEFT significantly outperforms VisProg and ViperGPT on the CLEVR transfer tasks. The LLM-based decomposition frameworks often fail to output executable code for complex queries,; in addition, the accuracy of execution by the APIs is low. For example, on the CLEVR-RPM task, only 0.40 of queries were executed successfully by ViperGPT, with accuracy of 0.10 out of executable queries. As these methods execute programs with a constrained set of pre-defined

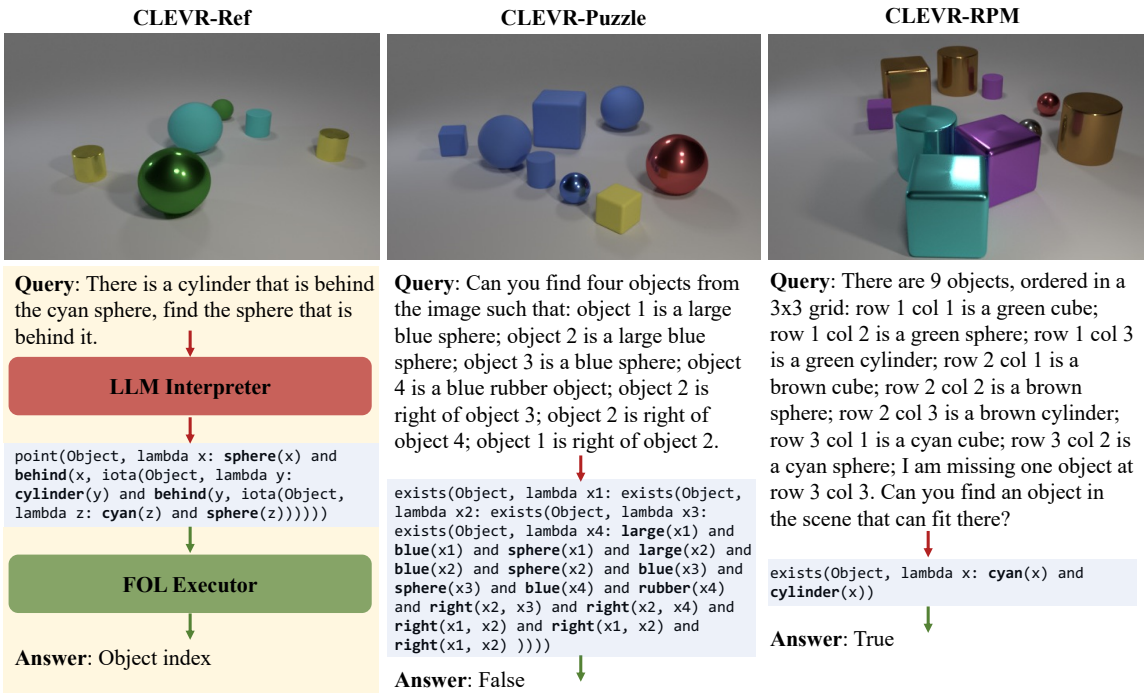


Figure 8-4: **Illustration of three zero-shot generalization tasks.** As seen in the CLEVR-Ref example, LEFT's domain-independent modules resolve the language into first-order logic and generate the answer.

	CLEVR-Ref	CLEVR-Puzzles	CLEVR-RPM
LEFT + GT programs	1.00	0.92	1.00
LEFT + LLM programs	**0.94**	**0.75**	**0.87**
VisProg [Gupta and Kembhavi, 2023]	0.35	0.27	0.51
ViperGPT [Surís *et al.*, 2023]	0.08	0.34	0.04
OpenFlamingo 4-shot [Awadalla *et al.*, 2023]	N/A	0.54	0.52
OpenFlamingo 8-shot	N/A	0.57	0.52

Table 8.6: **Zero-shot generalization of LEFT to three unseen tasks.** LEFT recomposes domain-independent programs and learned concepts with the LLM interpreter for flexible transfer.

functions, the accuracy for executing undefined relations, such as *to the right of*, which is learned in LEFT, is low. In addition, for complex domains such as 3D and temporal human motion sequences, it is unclear what pre-defined models should be used for these inference-only frameworks.

For comparison to general vision-language models, we evaluate OpenFlamingo [Awadalla *et al.*, 2023], an open-sourced version of Flamingo [Alayrac *et al.*, 2022]. OpenFlamingo is a VL model backed by a CLIP vision encoder [Radford *et al.*, 2021] and a LLAMA language encoder [Touvron *et al.*, 2023]. On both 4- and 8-shot variants, LEFT outperforms OpenFlamingo significantly. Notably, OpenFlamingo cannot predict answers for CLEVR-Ref, which requires an object index as output.

8.4 Related Work

LLM-based decomposition frameworks. Our framework integrates neural networks, logic reasoning, and large language models for commonsense reasoning. The first group of related literature studies LLM-based approaches. Inspired by the success of LLMs [Brown *et al.*, 2020], many recent works have proposed frameworks that leverage LLMs to decompose text-based tasks into a sequence of API calls to existing models [Cheng *et al.*, 2023; Dohan *et al.*, 2022; Beurer-Kellner *et al.*, 2023; Zelikman *et al.*, 2023]. Compared to LEFT, these methods run inference only with LLMs and API models, and are limited to the language domain, without learning grounding to modalities. For example, LLMs may be able to reason about categories of objects inferred from language, but it cannot recognize the candidate objects from the current scenes, or generate robotics movements to move the object. Works such as Gupta and Kembhavi [2023] and Surís *et al.* [2023] execute programs on images, but assume API access to a set of available modules with no further training. By contrast, LEFT learns to ground modular concepts across different domains by training concept embeddings for each modality, not requiring any pre-defined and pre-trained modules.

General vision-language models. In recent years, unified vision-language models (VLMs) have demonstrated success in learning on multimodal domains [Cho *et al.*, 2021; Tsimpoukelli *et al.*, 2021; Wang *et al.*, 2022; Alayrac *et al.*, 2022]. As a representative example, Flamingo [Alayrac *et al.*, 2022] leverages powerful pre-trained vision and language models to achieve few-shot learning on language generation tasks, interleaving images, videos, and text, then outputting associated texts. However, these unified VLMs, and broadly end-to-end methods, are still limited in its application in different domains. In addition, in practice, they struggle with more complex unseen tasks. By contrast, LEFT performs well across domains,

and can generalize to new challenging tasks that it has never seen.

Neuro-symbolic methods. Neuro-symbolic approaches that combine programs with neural networks have shown success in a variety of visual reasoning domains, with strong data efficiency and generalization abilities from their modular designs. Neuro-symbolic VQA [Yi *et al.*, 2018] proposed symbolic program execution for the question-answering task, and in Chapter 6, we have proposed NS-CL to further improved the training paradigm by removing the requirement for dense supervision. Inspired by the success in 2D image domains [Andreas *et al.*, 2016a; Johnson *et al.*, 2017b; Han *et al.*, 2019; Hudson and Manning, 2019b; Li *et al.*, 2020a; Chen *et al.*, 2021c; Mao *et al.*, 2021b], neuro-symbolic methods have been introduced for grounding in 3D scenes [Achlioptas *et al.*, 2020; Hong *et al.*, 2022; Hsu *et al.*, 2023b], reasoning in temporal sequences [Chen *et al.*, 2021d; Endo *et al.*, 2023], and robotic manipulation [Wang *et al.*, 2023b; Mao *et al.*, 2022]. However, these neuro-symbolic works require pre-defined domain-specific languages, such that task instructions are parsed into a constrained set of programs, and each program is then manually implemented in code. Therefore, it is challenging to incorporate commonsense reasoning into them, or generalize to different domains. In contrast, LEFT retains all the benefits of neuro-symbolic learning, while proposing a universal language for all domains. LEFT requires neither any domain-specific definition nor domain-specific program examples. It only leverages structures encoded through minimal prompting examples of first-order logic usage for the LLM; hence, it is generally applicable across domain.

8.5 Discussion

We have presented the Logic-Enhanced Foundation Model (LEFT), a unified concept learning and reasoning framework that flexibly learns concepts across different domains, and generalizes reasoning to unseen, complex tasks. Our framework combines foundation models and first-order logic executors as domain-independent reasoning modules, and learnable grounding modules for specific domains. LEFT represents a step towards modeling general reasoning capability in a rightfully grounded way.

Our framework currently does not model the interaction between the language and perception. Instead, LEFT relies on context-dependent grounding modules which learn from data to resolve ambiguous language. Future directions for LEFT include building in pragmatic reasoning frameworks, modelling speaker intents, and leveraging feedback from perception execution.

Chapter 9

Visual Concept Lexicon

In Chapters 6–8, we have focused on learning visual concepts from paired visual observations and language. In this chapter, we explore the reverse direction: how visual concept learning can facilitate language learning, such as acquiring the grammar rules of a new language from grounded experiences.

Concretely, we present Grammar-Based Grounded Lexicon Learning (G2L2), a lexicalist approach toward learning a compositional and grounded meaning representation of language from grounded data, such as paired images and texts. At the core of G2L2 is a collection of lexicon entries, which map each word to a tuple of a syntactic type and a neuro-symbolic semantic program. For example, the word *shiny* has a syntactic type of *adjective*; its neuro-symbolic semantic program has the *symbolic* form $\lambda x.\mathit{filter}(x, \textbf{SHINY})$, where the concept **SHINY** is associated with a *neural network* embedding, which will be used to classify shiny objects. Given an input sentence, G2L2 first looks up the lexicon entries associated with each token. It then derives the meaning of the sentence as an executable neuro-symbolic program by composing lexical meanings based on syntax. The recovered meaning programs can be executed on grounded inputs. To facilitate learning in an exponentially-growing compositional space, we introduce a joint parsing and expected execution algorithm, which does local marginalization over derivations to reduce the training time. We evaluate G2L2 on two domains: visual reasoning and language-driven navigation. Results show that G2L2 can generalize from small amounts of data to novel compositions of words.

Conceptually, this framework unifies visual concept learning (NS-CL) with word learning, enabling the joint acquisition of syntax and semantic forms of words and sentences. This addresses a key question in the broader concept learning framework: where do concepts come from? The answer, in our framework, is through the acquisition of word meanings.

The material of this chapter has been previously published in Mao *et al.* [2021b].

9.1 Introduction

Human language learning suggests several desiderata for machines learning from language. Humans can learn grounded and compositional representations for novel words from few examples. These representations are grounded on contexts, such as visual perception. We also know how these words relate with each other in composing the meaning of a sentence. *Syntax*—the structured, order-sensitive relations among words in a sentence—is crucial in humans' learning and compositional abilities for language. According to *lexicalist* linguistic theories [Pollard and Sag, 1994; Steedman, 2000; Bresnan *et al.*, 2016], syntactic knowledge involves a small number of highly abstract and potentially universal combinatory rules, together with a large amount of learned information in the lexicon: a rich syntactic type and meaning representation for each word.

Fig. 9-1 illustrates this idea in a visually grounded language acquisition setup. The language learner looks at a few examples containing the novel word *shiny* (Fig. 9-1a). They also have a built-in, compact but universal set of combinatory grammar rules (Fig. 9-1b) that describes how the semantic program of words can be combined based on their syntactic types. The learner can recover the syntactic type of the novel word and its semantic meaining. For example, *shiny* is an adjective and its meaning can be grounded on visually shiny objects in images (Fig. 9-1c). This representation supports the interpretation of novel sentences in a novel visual context (Fig. 9-1d).

In this chapter, we present Grammar-Based Grounded Lexicon Learning (G2L2), a neuro-symbolic framework for grounded language acquisition. At the core of G2L2 is a collection of grounded lexicon entries. Each lexicon entry maps a word to (i) a syntactic type, and (ii) a neuro-symbolic semantic program. For example, the lexicon entry for the English word *shiny* has a syntactic type of *objset/objset*: it will compose with another constituent of type *objset* on its right, and produces a new constituent of syntactic type *objset*. For example, in Fig. 9-1d, the word *shiny* composes with the word *cube* and yields a new constituent of type *objset*. The neuro-symbolic semantic program for *shiny* has the form $\lambda x.\mathit{filter}(x, \textbf{SHINY})$, where **SHINY** is a concept *automatically* discovered by G2L2 and associated with a learned vector embedding for classifying shiny objects. G2L2 parses sentences based on these grounded lexicon entries and a small set of combinatory categorial grammar [CCG; Steedman, 1996] rules. Given an input question, G2L2 will lookup the lexicon entries associated with each token, and compose these lexical semantic programs based on their syntactic types.

G2L2 takes a lexicalist approach toward grounded language learning and focuses on data efficiency and compositional generalization to novel contexts. Inspired by lexicalist linguistic theories, but in contrast to neural network-based end-to-end learning, G2L2 uses

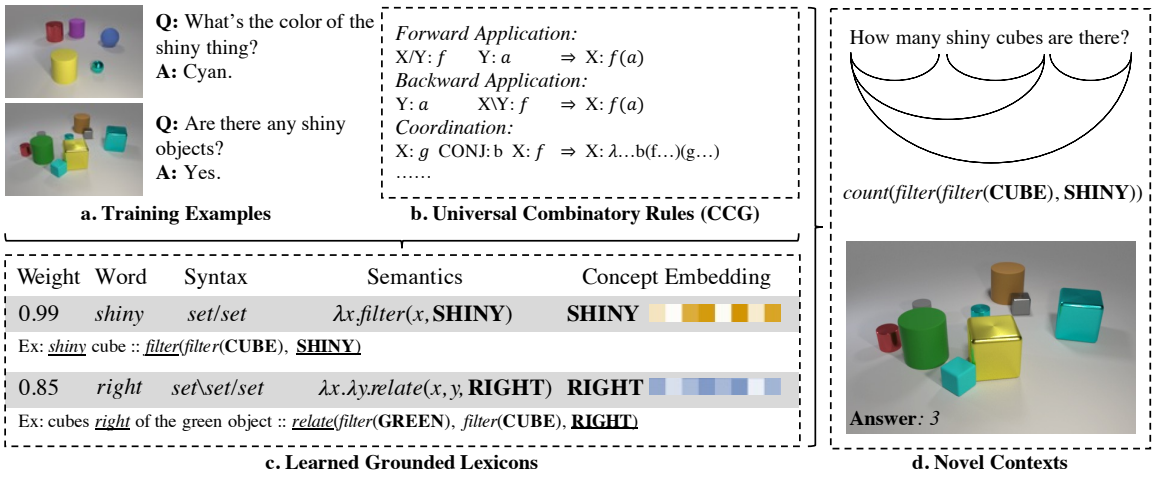


Figure 9-1: **Illustration of G2L2.** G2L2 learns from grounded language data, for example, by looking at images and reading parallel question–answer pairs. It learns a collection of grounded lexicon entries comprised of weights, syntax types, semantics forms, and optionally, grounded embeddings associated with semantic concepts. These lexicon entries can be used to parse questions into programs.

a compact symbolic grammar to constraint how semantic programs of individual words can be composed, and focuses on learning the lexical representation. This approach brings us strong data efficiency in learning new words, and strong generalization to new word compositions and sentences with more complex structures.

We are interested in jointly learning these neuro-symbolic grounded lexicon entries and the grounding of individual concepts from grounded language data, such as by simultaneously looking at images and reading parallel question–answer pairs. This is particularly challenging because the number of candidate lexicon entry combinations of a sentence grows exponentially with respect to the sentence length. For this reason, previous approaches to lexicon learning have either assumed an expert-annotated set of lexical entries [Zettlemoyer and Collins, 2009] or only attempted to learn at very small scales [Gauthier *et al.*, 2018]. We address this combinatory explosion with a novel joint parsing and *expected execution* mechanism, namely CKY-E2, which extends the classic CKY chart parsing algorithm. It performs local marginalization of distributions over sub-programs to make the search process tractable.

In sum, this chapter makes three specific contributions. First, we present the neuro-symbolic G2L2 model that learns grounded lexical representations without requiring annotations for the concepts to be learned or partial word meanings; it automatically recovers underlying concepts in the target domain from language and experience with their groundings. Second, we introduce a novel expected execution mechanism for parsing in model training, to facilitate search in the compositional grammar-based space of meanings. Third,

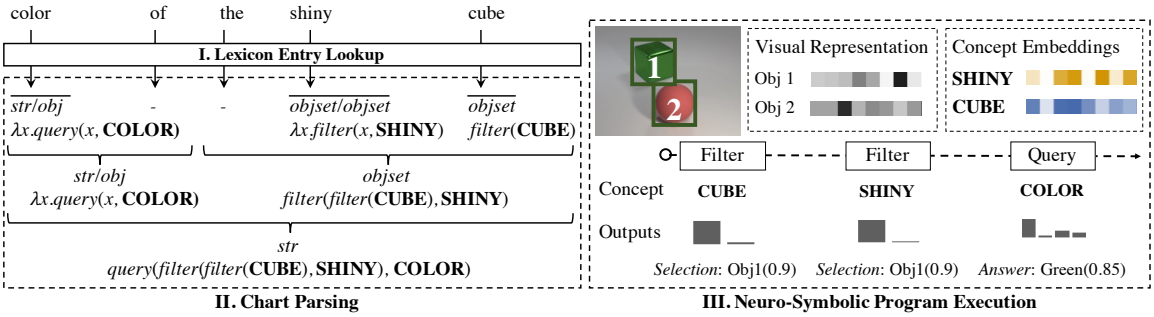


Figure 9-2: **Parsing and program execution of G2L2.** G2L2 parses the input sentence into an executable neuro-symbolic program by first (I) lookup the lexicon entry associated with each word, followed by (II) computes the most probable parsing tree and the corresponding tree with a chart parsing algorithm. The derived program can be grounded and executed on an image with a neuro-symbolic reasoning process (III).

through systematic evaluation on two benchmarks, visual reasoning in CLEVR [Johnson *et al.*, 2017a] and language-driven navigation in SCAN [Lake and Baroni, 2018], we show that the lexicalist design of G2L2 enables learning with strong data efficiency and compositional generalization to novel linguistic constructions and deeper linguistic structures.

9.2 Grammar-Based Grounded Lexicon Learning

Our framework, Grammar-Based Grounded Lexicon Learning (G2L2) learns grounded lexicons from cross-modal data, such as paired images and texts. Throughout this section, we will be using the visual reasoning task, specifically visual question answering (VQA) as the example, but the idea itself can be applied to other tasks and domains, such as image captioning and language-driven navigation.

G2L2 learns from a collection of VQA data tuples, containing an image, a question, and an answer to the question. In G2L2, each word type w is associated with one or multiple lexical entries, comprised of their syntactic types and semantic programs. Given the input question, G2L2 first looks up the lexicon entries associated with each individual token in the sentence (Fig. 9-2I). G2L2 then uses a chart parsing algorithm to derive the programmatic meaning representation of the entire sentence by recursively composing meanings based on syntax (Fig. 9-2II). To answer the question, we execute the program on the image representation (Fig. 9-2III). During training, we compare the answer derived from the model with the groundtruth answer to form the supervision for the entire system. No additional supervision, such as lexicon entries for certain words or concept labels, is needed.

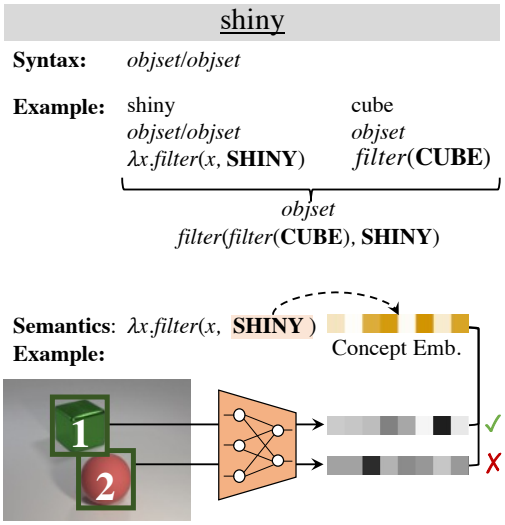


Figure 9-3: **Grounded lexicon in G2L2.** Each word is associated with a grounded lexicon, comprised of its syntactic type and a neuro-symbolic semantic program.

9.2.1 Grounded Lexicon

At a high-level, G2L2 follows the combinatory categorical grammar [CCG; Steedman, 1996] formalism to maintain lexicon entries and parse sentences. Illustrated in Fig. 9-3, Each word w (e.g., *shiny*) is associated with one or multiple entries. Each entry $e_w^{(i)}$ is a tuple comprised of a syntax type $syn_w^{(i)}$ (e.g., *objset/objset*), and a semantic meaning form $sem_w^{(i)}$ (e.g., $\lambda x.filter(x, \textbf{SHINY})$). $sem_w^{(i)}$ is a symbolic program represented in a typed domain-specific language (DSL) and can be executed on the input image. Some programs contain concepts (in this case, **SHINY**) that can be visually grounded.

Typed domain specific language. G2L2 uses a DSL to represent word meanings. For the visual reasoning domain, we use the CLEVR DSL [Johnson *et al.*, 2017a]. It contains object-level operations such as selecting all objects having a particular attribute (e.g., the shiny objects) or select all objects having a specific relationship with a certain object (e.g., the objects left of the cube). It also supports functions that respond to user queries, such as counting the number of objects or query a specific attribute (e.g., shape) of an object. The language is typed: most functions takes a set of objects or a single object as their inputs, and produce another set of objects. For example, the operation *filter* has the signature $filter(objset, concept) \rightarrow objset$ and returns all objects that have *concept* (e.g., all *shiny* objects) in the input set.

Syntactic types. There are two types of syntactic types in G2L2: primitive and complex.* The primitive types are defined in the typed domain specific language (e.g., *objset*, *int*). A complex type, denoted as X/Y or X\Y, is a functor type that takes an argument of type Y and returns an object of type X. The direction of the slash indicates word order: for X/Y, the argument Y must appear on the right, whereas in X\Y, it must appear on the left. Note that X and Y can themselves be complex types, which allows us to define functor types with multiple arguments, such as (X\Y)/Z, or even functors with functor arguments (e.g., (X\Y)/(Z/Z)).

In G2L2, the semantic type of a word meaning (in the DSL) together with a set of directional and ordering settings for its arguments (reflecting how the word and its arguments should be linearized in text) uniquely determines the word's syntactic type. For example, the syntactic type for word *shiny* is *objset/objset*. It first states that *shiny* acts as a function in meaning composition, which takes a subprogram that outputs a set of objects (e.g., *filter*(**CUBE**)) as its argument, and produces a new program whose output is also a set of objects, in this case, *filter*(*filter*(**CUBE**), **SHINY**) Second, it states the direction of the argument, which should come from its right.

Neuro-symbolic programs. Some functions in the DSL involves concepts that will be grounded in other modalities, such as the visual appearance of an object and their spatial relationships. Taking the function *filter* as an example, its secondary argument *concept* should be associated with the visual representation of objects. In G2L2, the meaning of each lexicon entry may involve one more constants (called "concepts") that are grounded on other modalities, possibly via deep neural embeddings. In the case of *shiny*: $\lambda x.$*filter*$(x,$ **SHINY**$)$. The concept **SHINY** is associated with a vector embedding in a joint visual-semantic embedding space, following Kiros *et al.* [2014]. During program execution, we will be comparing the embedding of concept **SHINY** with object embeddings extracted from the input image, to filter out all *shiny* objects.

Lexicon learning. G2L2 learns lexicon entries in the following three steps. (i) First, we enumerate all possible semantic meaning programs derived from the DSL. For example, in the visual reasoning domain, a candidate program is $\lambda x.$*filter*$(x,$ **?**$)$, where **?** denotes a concept argument. When we try to associate this lexicon entry to the word *shiny*, the program is instantiated as $\lambda x.$*filter*$(x,$ **SHINY**$)$, where **SHINY** is a new concept associated with a vector embedding. Typically, we set a maximum number of arguments for each program and constrain its depth. We explain how we set these hyperparameters for different domains in Appendix I.3. (ii) Next, for programs that have a primitive type, we use its semantic

*In some domains we also use conjunctions (*CONJ*) in the coordination rule.

type as the syntactic type (e.g., *objset*). For programs that are functions with arguments, we enumerate possible argument ordering of the arguments. For example, the program $\lambda x.filter(x, \textbf{SHINY})$ has two candidate syntactic types: *objset/objset* (the argument is on its right in language) and *objset\objset* (the argument is on its left). (iii) Finally, we associate each candidate lexicon entry with a learnable scalar weight $\tau(\cdot)$. It is typical for a single word having tens or hundreds of candidate entries, and we optimize these lexicon entry weights in the training process. In practice, we assume no lexical ambiguity, i.e., *each word type has only one lexical entry*. Thus, the ambiguity of parsing only comes from different syntactic derivation orders for the same lexical entries. This also allows us to prune lexicon entries that do not lead to successful derivations during training.

9.2.2 Program Execution

Any fully grounded programs (i.e., programs without unbound arguments) can be executed based on the image representation. We implement the neuro-symbolic concept learner presented in Chapter 6 as our differentiable program executor. To quickly recap, NS-CL consists of a collection of deterministic functional modules to realize the operations in the DSL. It represents execution results in a "soft" manner: in the visual reasoning domain, a set of objects is represented as a vector mask m of length N, where N is the number of objects in the scene. Each element, m_i can be interpreted as the probability that object i is in the set. For example, the operation $\lambda x.filter(x, \textbf{SHINY})$ receives an input mask m and produces a mask m' that selects all shiny objects in the input set. The computation has two steps: (i) compare the vector embedding of concept **SHINY** with all objects in the scene to obtain a mask $m^{(\textbf{SHINY})}$, denoting the probability of each object being *shiny*; (ii) compute the element-wise multiplication $m' = m \odot m^{\textbf{SHINY}}$, which can be further used as the input to other functions. In NS-CL, the execution result of any program is fully differentiable w.r.t. the input image representation and concept embeddings (e.g., **SHINY**).

9.2.3 Joint Chart Parsing and Expected Execution (CKY-E2)

G2L2 extends a standard dynamic programming algorithm for chart parsing (i.e., the CKY algorithm [Kasami, 1966; Younger, 1967; Cocke, 1969]) to compose sentence meaning from lexical meaning forms, based on syntax. Denote w_i as the input word sequence. e_i^j the j-th lexicon entry associated with word w_i, and $\tau(e_i^j)$ the corresponding weight. Consider all possible derivation of the question $\{derivation_k\}$, $k = 1, 2, \ldots$. We define the following context-free probability distribution of derivations as:

Algorithm 6 The CKY-E^2 algorithm.

Input: w_i: the input sentence; L: sentence length; e_i^j: the j-th lexicon entry associated with word w_i; $\tau(e_i^j)$: lexicon weights.

Output: exe_k the execution result of the all possible derivations and their weights $\tau(exe_k)$.

for $i \leftarrow 0$ to $L - 1$ **do**
  Initialize $dp[i, i+1]$ with lexicon entries e_i^* and weights $\tau(e_i^*)$
for *length* $\leftarrow 1$ to L **do**
  for *left* $\leftarrow 0$ to $L -$ *length* **do**
    right $\leftarrow$ *left* + *length*
    $dp[left, right] \leftarrow$ empty list
    for $k \leftarrow$ *left* $+ 1$ to *right* $- 1$ **do**
      Try to combine nodes in $dp[left, k]$ and $dp[k, right]$
      Append successful combination to $dp[left, right]$
    ExpectedExecution($dp[left, right]$)
procedure ExpectedExecution(a: a list of derivations)
  while $\exists x, y \in a$ are identical except for subtrees of the same type **do**
    Create z from x and y by computing the expected execution results for non-identical subtrees
    $\tau(z) \leftarrow \tau(x) + \tau(y)$
    Replace x and y in a with z

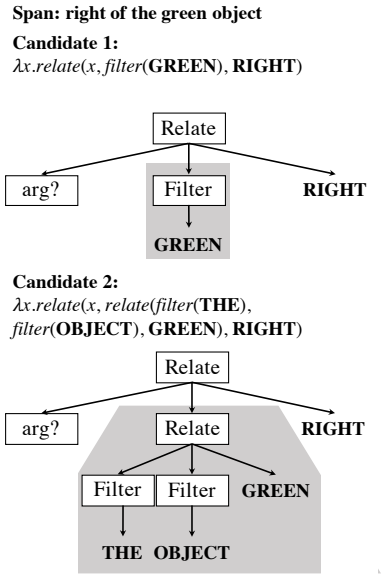


Figure 9-4: An illustrative example of two semantic programs that can be merged by computing the expected execution results of two subtrees (highlighted in gray). Both subtrees outputs a vector of scores indicating the objects being selected.

$$p(\textit{derivation}_k) \propto \exp\left(\sum_{e \in \textit{derivation}_k} \tau(e)\right).$$

That is, the probability is exponentially proportional to the total weights $\tau(e)$ of all lexicon entries $e \in \textit{derivation}_k$ used by the specific derivation.

A straightforward implementation to support joint learning of lexicon weights τ and neural modules (e.g., $\textit{filter}(x, \textbf{SHINY})$), is to simply execute all possible derivations on the input image, and compare the answer with the groundtruth. However, the number of possible derivations grows exponentially as the question length, making such computation intractable. For example, in SCAN [Lake and Baroni, 2018], each word has 178 candidate lexicons, and the number of lexicon combination of a sentence with 5 words will be $178^5 \approx 10^{11}$. To address this issue, we introduce the idea of expected execution, which essentially computes the "expected" execution result of all possible derivations. We further accelerate this process by taking local marginalization.

Our CKY-E2 algorithm is illustrated in Algorithm 6. It processes all spans $[\textit{left}, \textit{right})$ sequentially ordered by their length. The composition for derivations of $[\textit{left}, \textit{right})$ has two stages. First, it enumerates possible split point k and tries to combine the derivation of $[\textit{left}, k)$ and $[k, \textit{right})$. This step is identical to the standard CKY parsing algorithm. Next, if there are two derivations x and y of span $[i, j)$, whose program structures are identical except for subtrees that can be partially evaluated (i.e., does not contain any unbounded arguments), we will compress these two derivations into one, by marginalizing the execution result for that subtree.

See the example from Fig. 9-4. Two programs have the identical structure, except for the second argument to the outer-most *relate* operation. However, these sub-trees, highlighted in gray, can be partially evaluated on the input image, and both of them output a vector of scores indicating the objects being selected. Denote τ_1 and τ_2 as the weight associated with two derivations, and v_1 and v_2 the partial evaluation results (vectors) for two subtrees. We will replace these two candidate meaning form with z:

$$z := \lambda x.\textit{relate}(x, v', \textbf{RIGHT}), \quad \text{where } v' := \frac{\exp(\tau_1)v_1 + \exp(\tau_2)v_2}{\exp(\tau_1) + \exp(\tau_2)}, \tau(z) := \tau_1 + \tau_2.$$

We provide additional running examples of the algorithm in Appendix I.2.1.

Complexity. Intuitively, once we have determined the semantics of a constituents in the question, the actual concrete meaning form of the derivation does not matter for future program execution, if the meaning form can already be partially evaluated on the input image.

This joint parsing and expected execution procedure significantly reduces the exponential space of possible parsing to a polynomial space w.r.t. the number of possible program layouts that can not be partially evaluated, which, in practice, is small. The complexity of CKY-E2 is polynomial with respect to the length L of the sentence, and M the number of candidate lexicon entries. More specifically, $O(L^3M)$, where $O(L^3)$ comes from the chart parsing algorithm, and the number of derivations after the expected execution procedure is $O(M)$. This result is obtained by viewing the maximum arity for functor types being a constant (e.g., 2). Intuitively, for each span, all possible derivations associated with this span can be grouped into 4 categories: derivations of a primitive type, derivations of a 1-ary functor type, derivations of a 2-ary functor type, and derivations of a 2-ary functor type, with one argument bound. All these numbers grow linearly w.r.t. M. For detailed analysis please refer to Appendix I.2.2.

Correctness. One can theoretically prove that, if all operations in the program layout are commutative with the expectation operator, i.e., if $\mathbb{E}([f(x)] = f(\mathbb{E}[x])$, our CKY-E2 produces exact computation of the expected execution result. These operations include, tensor addition, multiplication (if tensors are independent), and concatenation, which cover most of the computation we will do in neuro-symbolic program execution. For example, for *filter*, taking the expectation over different inputs before doing the filtering is the same as taking the expectation over the filter results of different inputs. However, there are operations such as quantifiers whose semantics are not commutative with the expectation operator. In practice, it is possible to still use the expected expectation framework to approximate. We leave the application of other approximated inference techniques as future work. We provide proofs and its connections with other formalisms in Appendix I.2.2.

9.2.4 Learning

Our model, G2L2, can be trained in an end-to-end manner, by looking at images and reading paired questions and answers. We denote ℓ as a loss function that compares the output of a program execution (e.g., a probability distribution over possible answers) and the groundtruth. More precisely, given all possible derivations $derivation_k$, the image representation I, the answer A, and the executor $\mathcal{E}(\cdot, I)$, we optimize all parameters by minimizing the loss $\mathcal{L}$:

$$\mathcal{L} = \sum_k \left(p(derivation_k) \cdot \ell\left(\mathcal{E}(derivation_k, I), A\right)\right).$$

In practice, we use gradient-based optimization for both the neural network weights in concept grounding modules and the lexicon weights τ.

Model	Prog?	Concept?	Standard		Compositional Generalization			Depth
			10%	100%	*purple*	*right of*	*count*	
MAC [Hudson and Manning, 2018]	N/N	N/N	85.39	98.61	97.14	90.85	54.87	77.40
TbD-Net [Mascharka *et al.*, 2018]	Y/Y	N/N	44.52	98.04	89.57	49.92	63.37	53.13
NS-VQA [Yi *et al.*, 2018]	Y/Y	Y/Y	**98.57**	98.57	95.52	**99.80**	81.81	50.45
NS-CL (Chapter 6)	Y/N	Y/N	98.51	**98.91**	**98.02**	99.01	18.88	81.60
G2L2 (ours)	Y/N	Y/N	98.11	98.25	97.82	98.59	**96.76**	**98.49**

Table 9.1: **Results on CLEVR.** Our model achieves a comparable results with state-of-the-art approaches on the standard training-testing split. It significantly outperforms all baselines on generalization to novel word compositions and to sentences with deeper structures. The best number in each column is bolded. The second column indicates whether the model uses program-based representation of question meaning and whether it needs program annotation for training questions. The third column indicates whether the model explicitly models individual concepts and whether it needs concept annotation for objects during training.

9.3 Experiment

We evaluate G2L2 on two domains: visual reasoning in CLEVR [Johnson *et al.*, 2017a] and language-driven navigation in SCAN [Lake and Baroni, 2018]. Beyond the grounding accuracy, we also evaluate the compositional generalizability and data efficiency, comparing G2L2 with end-to-end neural models and modular neural networks.

9.3.1 Visual Reasoning

We first evaluate G2L2 on the visual reasoning tasks in the CLEVR domain [Johnson *et al.*, 2017a], where the task is to reason and answer questions about images. In our study, we use a subset of CLEVR dataset, which does not include sentences that involve coreference resolution, and words with multiple meanings in different contexts. We add additional information on how we filter the dataset in Appendix I.3.

Setup. Instead of using manually defined heuristics for curriculum learning or self-paced learning as in previous works [Chapter 6; Li *et al.*, 2020b], we employ a curriculum learning setup that is simply based on sentence length: we gradually add longer sentences into the training set. This helps the model to learn basic words from very short sentences (6 words), and use the acquired lexicon to facilitate learning longer sentences (20 words). Since CLEVR does not provide test set annotations, for all models, we held out 10% of the training data for model development and test them on the CLEVR validation split.

Baselines. We compare G2L2 with 4 baselines. (1) MAC [Hudson and Manning, 2018] is an end-to-end approach based on attention. (2) TbD-Net [Mascharka *et al.*, 2018] uses a pre-trained semantic parser to parse the question into a symbolic program, and executes the program with a neural module network [Andreas *et al.*, 2016b]. (3) Similarly, NS-VQA [Yi *et al.*, 2018] also parses the question into a symbolic program. It also extracts an abstract scene representation with pre-trained neural recognition models [He *et al.*, 2017]. It executes the program based on the abstract scene representation. Both of the approaches require additional supervision for training the semantic parser, and NS-VQA requires additional annotation for training the visual recognition model. (4) NS-CL (Chapter 6) jointly learns a neural semantic parser and concept embeddings by looking at images and reading paired questions and answers. It requires the annotation for all concepts in the domain (e.g., colors and shapes). In contrast, G2L2 can *automatically* discover visual concepts from texts.

Results. Table 9.1 summarizes the results. We consider any model that performs in the 95–100 range to have more or less solved the task. Small differences in numeric scores in this range, such as the fact that NS-CL outperforms our model on the "purple" generalization task by 0.2%, are less important than the fact that our model far outperforms all competitors on "count" compositional generalization and the "depth" generalization task, both of which all competitor models are far from solving.

We first compare different models on the **standard** training-testing split. We train different models with either 10% or 100% of the training data and evaluate them on the validation set. Our model achieves a comparable performance in terms of its accuracy and data efficiency.

Next, we systematically build three **compositional generalization** test splits: *purple*, *right of*, and *count*. The detailed setups and examples for these splits are provided in Appendix I.3. Essentially, we remove 90% of the sentences containing the word *purple*, the phrase *right*, and *counting operations*, such as *how many ...?* and *what number of ...?*. We only keep sentences up to a certain length (6 for purple, 11 for right, and 8 for count). We make sure that each use case of these words appear in training questions. After training, we test these models on the validation set with questions containing these words. Overall, our model G2L2 outperforms all baselines on all three generalization splits. In particular, it significantly outperforms other methods on the *count* split. The *count* split is hard for other method because it requires model to generalize to sentences with deeper structures, for example, from "*how many red objects are there?*" to "*how many red objects are right of the cube?*" Note that, during training, all models have seen example use of similar structures such as "*what's the shape of the red object*" and "*what's the shape of the red object right of the cube?*"

Finally, we test generalization to sentences with deeper structures (**depth**). Specifically, we define the "hop number" of a question as the number of intermediate objects being referred to in order to locate the target object. For example, the "hop number" of the question "*how many red objects are right of the cube?*" is 1. We train different models on 0-hop and 1-hop questions and test them on 2-hop questions. Our model strongly outperforms all baselines.

The results on the **compositional generalization** and **depth** splits yield two conclusions. First, disentangling grounded concept learning (associating words onto visual appearances) and reasoning (e.g., filtering or counting subsets of objects in a given scene) improves data efficiency and generalization. On CLEVR, neuro-symbolic approaches that separately identify concepts and perform explicit reasoning (NS-VQA, NS-CL and G2L2) consistently generalize better than approaches that do not (MAC, TbD). The comparison between TbD and NS-VQA is informative: TbD fails on the "right of" task even in the case where the semantic parser is providing correct programs, while NS-VQA, which uses the same parser but explicitly represents compositional symbolic concepts for reasoning, succeeds in this task. Crucially, of the three neuro-symbolic methods, G2L2 achieves strong performance with less domain-specific knowledge than other methods: NS-VQA needs groundtruth programs; NS-CL needs the concept vocabulary; G2L2 requires neither. Second, our model is the only one to perform well on the hardest "out-of-sample" generalization tests: holding out "count" and generalizing to deeper embeddings. The other, easier generalization tests all have close neighbors in the training set, differing by just one word. In contrast, the length, depth and "count" tests require generalizing to sentences that differ in multiple words from any training example. They appear to require – or at least benefit especially well from – G2L2 's lexical-grammatical approach to capturing meaning of complex utterances, with explicit constituent-level (as opposed to simply word-level) composition. We also provide in-depth analysis for the behavior of different semantic parsing models in Appendix I.3

9.3.2 Language-driven Navigation

The second domain we consider is language-driven navigation. We evaluate models on the SCAN dataset [Lake and Baroni, 2018]: a collection of sentence and navigational action sequence pairs. There are 6 primitive actions: *jump*, *look*, *walk*, *run*, *lturn*, and *rturn*, where an instruction *turn left twice and run* will be translated to *lturn lturn run*. All instructions are generated from a finite context-free grammar, so that we can systematically construct train-test splits for different types of compositional generalizations.

Setup. We use a string-editing domain-specific language (DSL) for modeling the meaning of words in the SCAN dataset, of which the details can be found in Appendix I.3. At a

Model	Simple		Compositional Generalization		Length
	10%	100%	*jump*	*around right*	
seq2seq [Sutskever *et al.*, 2014]	0.93±0.05	0.99±0.01	0.00±0.00†	0.00±0.00†	0.15±0.02
Transformer [Vaswani *et al.*, 2017]	0.71±0.24	0.78±0.11	0.00±0.00	0.10±0.08	0.02±0.01
GECA [Andreas, 2020]	0.99±0.00	0.98±0.01	0.87±0.05†	0.82±0.11†	0.15±0.02
WordDrop [Guo *et al.*, 2020]*	0.56±0.02	0.62±0.02	0.52±0.02	0.70±0.06	0.18±0.01
SwitchOut [Wang *et al.*, 2018d]*	0.99±0.01	0.99±0.01	0.98±0.02	0.97±0.02	0.17±0.02
SeqMix [Guo *et al.*, 2020]*	–	–	0.98‡	0.89‡	–
recomb-2 [Akyürek *et al.*, 2021]	–	–	0.88±0.07†	0.82±0.08†	–
G2L2 (ours)	**1.00**±0.00	**1.00**±0.00	**1.00**±0.00	**1.00**±0.00	**1.00**±0.00

Table 9.2: **Results on SCAN.** Accuracies are averaged across 10 valid runs when applicable, ± denotes standard deviation. The best number in each column is bolded. †: results taken from [Akyürek *et al.*, 2021]; ‡: results taken from [Guo *et al.*, 2020]. Both paper have only presented results on the compositional generalization split. ∗: applied after GECA. The results for GECA are based on the released implementation by the authors. All the models are selected with respect to the accuracy on the training set.

high level, the model supports three primitive operations: constructing a new constant string (consisting of primitive operations), concatenating two strings, and repeating the input string for a number of times.

For G2L2, we generate candidate lexicons by enumerating functions in the string-editing DSL with up to 2 arguments and the function body has a maximum depth of 3. We also allow at most one of the argument being functor-typed, for example, $V\backslash V/(V\backslash V)$. To handle parsing ambiguities, we use two primitive syntax types S and V, while both of them are associated with the semantic type of *string*. In total, we have 178 candidate lexicon entries for each word.

Baselines. We compare G2L2 to seven baselines. (1) Seq2seq [Sutskever *et al.*, 2014] trains an LSTM-based encoder-decoder model. We follow the hyperparameter setups of [Lake and Baroni, 2018]. (2) Transformer [Vaswani *et al.*, 2017] is a 4-head Transformer-based autoregressive seq2seq model. We tuned the hidden size (i.e., the dimension of intermediate token representations) within {100, 200, 400}, as well as the number of layers (for both the encoder and the decoder) from {2, 4, 8}. Other methods are based on different data augmentation schemes for training a LSTM seq2seq model. Specifically, (3) GECA augments the original training splits using heuristic span recombination rules; (4) WordDrop [Guo *et al.*, 2020] performs random dropout for input sequence (while keeping the same label); (5) similarly, SwitchOut [Wang *et al.*, 2018d] randomly replaces an input token with a random token from the vocabulary; (6) SeqMix [Guo *et al.*, 2020] uses soft augmen-

tation techniques following [Zhang *et al.*, 2017], which composes an "weighted average" of different input sequences; (7) recomb-2 [Akyürek *et al.*, 2021] learns recombination and resampling rules for augmentation.

Results. We compare different models on three train-test splits (Table 9.2). In **Simple**, the training and test instructions are drawn from the same distribution. We compare the data efficiency of various models by using either 10% or 100% of the training data, and test them on the same test split. While all models can achieve a nearly-perfect accuracy with 100% training data, our model G2L2 shows advantage with only a small amount of data. Next, in **Compositional**, we have held out the sentences containing certain phrases, such as *jump* and *around right*. For these held-out phrases, only valid non-contextual examples containing them (i.e., *jump* in isolation and no example for *around right*) are available during training. During test, algorithms need to make systematical generalization of these phrases in novel contexts. Finally, in **Length**, all training examples have the action length less than or equal to 22, while that of a test example is up to 48. Our model consistently reach perfect performance in all considered settings, even on the cross-length generalization task where GECA does not help improve performance. These results are consistent with the conclusions we derived on the CLEVR dataset. Specifically, data-augmentation techniques for SCAN can solve simple generalization tests (e.g., *jump*, where tests all have close neighbors in the training set, differing by just one word) but not the hard ones (e.g., *length*, where test sentences can different in multiple words from any training examples).

Cases study. G2L2 is expressive enough to achieve perfect accuracy on the SCAN dataset: there exists a set of lexicon entries which matches the groundtruth in SCAN. However, our learning algorithm does not always converge on the correct lexicon, but when it fails, the failure can be identified based on training-set accuracy. So, we perform model selection based on the training accuracy for G2L2: after a sufficient number of epochs, if the model hasn't reached perfect accuracy (100%), we re-initialize the weights and train the model again. Our results show that, among 100 times of training, the model reaches 100% accuracy 74% of the time. For runs that don't have 100% accuracy, the average performance is 0.94.

Since G2L2 directly learns human-interpretable lexicon entries associated with each individual words, we can further inspect the failure cases made by it when the training accuracy does not converge to 0. We find that the most significant failure mode is the word *and* (e.g., *jump and run*) and *after* (e.g., *jump after run*). Both of them are treated as connectives in SCAN. Sometimes G2L2 fails to pick the syntax type $S\backslash V/V$ over the type $V\backslash V/V$. The entry $V\backslash V/V$ will succeed in parsing most cases (e.g., *jump and run*), except that it will introduce ambiguous parsing for sentences such as "*jump and run twice*": *jump*

and run twice vs. jump and run twice. Based on the definition of the SCAN, only the first derivation is valid. In contrast, using $S\backslash V/V$ resolves this ambiguity. Depending on the weight initialization and the example presentation order, G2L2 sometimes get stuck at the local optima of $V\backslash V/V$. However, we can easily identify this by the training accuracies—G2L2 is able to reach perfect performance on all considered splits by simply retraining with another random seed, therefore, we only select those with 100% training accuracy as valid models.

9.4 Related Work

Lexicalist theories. The lexicalist theories of syntax [Pollard and Sag, 1994; Steedman, 2000; Bresnan *et al.*, 2016] propose that 1) the key syntactic principles by which words and phrases combine are extremely simple and general, and 2) nearly all of the complexity in syntax can be attributed to rich and detailed lexical entries for the words in the language. For example, whereas the relationship between the active and passive voice, e.g., "Kim saw a balloon" versus "A balloon was seen by Kim", was treated in pre-lexicalist theories as a special syntactic rule converting between the sentences, in lexicalist theories this relationship derives simply from the knowledge that the passive participle for the verb "see" is "seen," which interacts with knowledge of other words to make both the active and passive forms of the sentence possible. In lexicalist theories, the problem for the language learner thus becomes a problem of learning the words in the language, not a problem of learning numerous abstract rule schemas. The combinatory categorial grammar [CCG; Steedman, 1996] framework we use is a well-established example of a lexicalist theory: there is a universal inventory of just three combinatory rules (Fig. 9-1a), but those rules can only be applied once richly specified lexical entries are learned for the words in a sentence. We believe that this lexicalist-theory approach is a particularly good fit to the problem of grounded language learning: the visual context provides clues to the word's meaning, and the word's grammatical behavior is tied closely to this meaning, making learning efficient.

Compositional generalization in NLP. Improving the compositional generalization of natrual langauge processing (NLP) systems have drawn great attention in recent years [Baroni, 2020]. Most of the recent approaches towards this goal are mostly built on deep learning-based models. There are two representative approaches: building structured neural networks with explicit phrase-based structures or segments [Socher *et al.*, 2013; Zhu *et al.*, 2015a; Tai *et al.*, 2015; Saqur and Narasimhan, 2020]; and using data augmentation techniques [Andreas, 2020; Guo *et al.*, 2020; Akyürek *et al.*, 2021]. However, these approaches either rely on additional annotation or pretrained models for phrase structure inference or require domain-specific heuristics in data augmentation. In contrast to both approaches, we

propose to use combinatory grammar rules to constrain the learning of word meanings and how they compose.

Neural latent trees. CKY-E2 is in spirit related to recent work using CKY-style modules for inducing latent trees. However, our model is fundamentally different from works on unsupervised constituency parsing [Kim *et al.*, 2019; Shi *et al.*, 2021] which use the CKY algorithm for inference over scalar span scores and those compute span representation vectors with CKY-style algorithms [Maillard and Clark, 2018; Drozdov *et al.*, 2019, *inter alia*]. Our key contribution is to introduce the expected execution mechanism, where each span is associated with weighted, compressed programs. Beyond enumerating all possible parsing trees as in [Maillard and Clark, 2018], G2L2 considers all possible programs associated with each span. Our expected execution procedure works for different types (object set, integer, etc.) and even functor types. This makes our approximation exact for linear cases and has polynomial complexity.

Grammar-based grounded language learning. There have also been approaches for learning grammatical structures from grounded texts [Shi *et al.*, 2019a; Zhao and Titov, 2020; Jin and Schuler, 2020; Artzi and Zettlemoyer, 2013; Ross *et al.*, 2018; Tellex *et al.*, 2011]. However, these approaches either rely on pre-defined lexicon entries [Artzi and Zettlemoyer, 2013], or only focus on inducing syntactic structures such as phrase-structure grammar [Shi *et al.*, 2019a]. Different from them, G2L2 jointly learns the syntactic types, semantic programs, and concept grounding, only based on a small set of combinatory grammar rules.

Grammar-based and grounded language learning have also been studied in linguistics, with related work to ours studying on how humans use grammar as constraints in learning meaning [Steedman, 2000] and how learning syntactic rules and semantic meanings in language bootstrap each other[Abend *et al.*, 2017; Taylor and Gelman, 1988]. However, most previous computational models have focused only on explaining small-scale lab experiments and do not address grounding in visual perception [Fazly *et al.*, 2010; Gauthier *et al.*, 2018]. In contrast, G2L2 is a neuro-symbolic model that integrates the combinatory categorial grammar formalism [Steedman, 1996] with joint perceptual learning and concept learning, to directly learn meanings from images and texts.

Neuro-symbolic models for language grounding. Integrating symbolic structures such as programs and neural networks has shown success in modeling compositional queries in various domains, including image and video reasoning [Hu *et al.*, 2017; Mascharka *et al.*, 2018], knowledge base query [Andreas *et al.*, 2016a], and robotic planning [Andreas *et al.*,

2017]. In this chapter, we use symbolic domain-specific languages with neural network embeddings for visual reasoning in images and navigation sequence generation, following NS-CL in Chapter 6. However, in contrasts to using neural network-based semantic parser as in the aforementioned papers, our model G2L2 focuses on learning grammar-based lexicon for compositional generalization in linguistic structures, such as novel word composition.

9.5 Discussion

In this chapter, we have presented G2L2, a lexicalist approach towards learning compositional and grounded meaning of words. G2L2 builts in a compact but potentially universal set of combinatory grammar rules and learns grounded lexicon entries from a collection of sentences and their grounded meaning, without any human annotated lexicon entries. The lexicon entries represent the semantic type of the word, the ordering settings for its arguments, as well as the grounding of concepts in its semantic program. To facilitate lexicon entry induction in an exponentially-growing space, we introduced CKY-E2 for joint chart parsing and *expected execution*.

Through systematic evaluation on both visual reasoning and language-driven navigation domains, we demonstrate the data efficiency and compositional generalization capability G2L2, and its general applicability in different domains. The design of G2L2 suggests several research directions. First, in G2L2 we have made strong assumptions on the context-independence of the lexicon entry as well as the application of grammar rules, the handling of linguistic ambiguities and pragmatics needs further exploration [Frank and Goodman, 2012]. Second, meta-learning models that can leverage learned words to bootstrap the learning of novel words, such as syntactic bootstrapping [Gauthier *et al.*, 2018], is a meaningful direction. Finally, future work may consider integrating G2L2 with program-synthesis algorithms [Ellis *et al.*, 2023] for learning of more generic and complex semantic programs.

Part III

Neuro-Symbolic Action Concepts

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 10

Neuro-Symbolic Skills

We begin Part III by introducing representations and reasoning algorithms for neuro-symbolic skills, which are relatively short-horizon "controllers" that can be executed by the robot to accomplish a certain goal (e.g., opening a drawer).

First, we introduce the notion of mechanisms, reusable manipulation strategies characterized as sequences of contact mode changes coupled with mechanism-specific samplers. From a single demonstration, a robot can extract the causal structure of contact interactions, refine them through self-play, and integrate the resulting mechanisms as macros within a task-and-motion planner. This approach compresses long planning horizons into symbolic units that can be efficiently recombined through reasoning (in particular, continuous constraint satisfaction as in Chapter 5) and planning (as tree search) to solve novel problems.

To overcome the limitations of mechanisms in generalizing to unseen objects, we develop MAGIC (Manipulation Analogies for Generalizable Intelligent Contacts), a framework for one-shot strategy transfer. MAGIC extracts contact waypoints from demonstrations and establishes functional correspondences on novel objects through a global-to-local analogy process: coarse shape alignment using pretrained visual features followed by fine-grained geometric matching based on local curvature. By reasoning directly about forceful affordances rather than memorizing object instances, MAGIC enables efficient generalization to diverse tools and object categories.

Finally, we extend the framework toward learning more complex contact constraints with KALM (Keypoint Abstraction using Large Models). KALM prompts large pre-trained vision-language models to propose candidate task-relevant keypoints and verifies their consistency across demonstrations, distilling them into compact, cross-instance abstractions. These keypoints serve both as object-relative reference frames and as inputs for keypoint-conditioned policy learning, enabling data-efficient imitation and robust generalization across new object poses and categories.

Overall, this chapter establishes a unified paradigm for neuro-symbolic composable skills:

mechanisms capture the symbolic structure of contact sequences, MAGIC enables analogy-based generalization across novel objects, and KALM grounds higher-level abstractions in automatically discovered keypoints. This framework enables agents to learn skills from as few as 1 to 10 demonstrations, and to apply them compositionally to solve longer-horizon manipulation problems, which we will dive deeper into in Chapter 11.

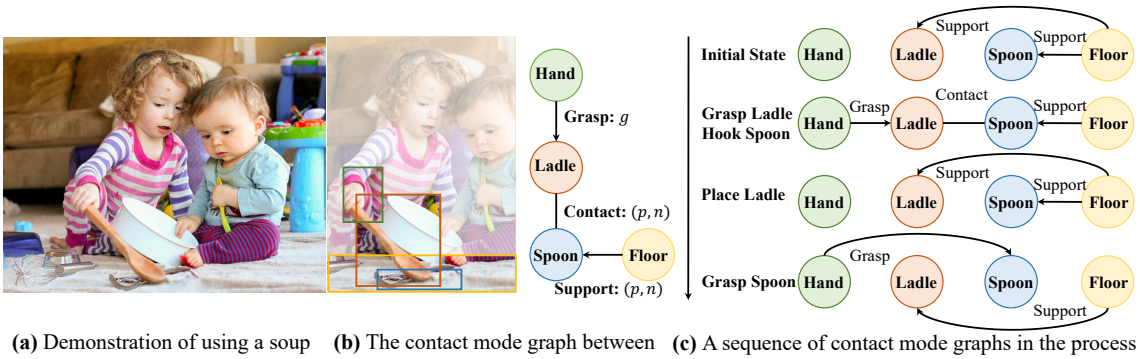
The material of this chapter has been previously published in Mao *et al.* [2023b], Liu *et al.* [2025], and Fang *et al.* [2025]. Yuyao Liu, Xiaolin Fang, Bo-Ruei Huang, and Jasmine Shone, in particular, contributed significantly to the materials presented in this chapter.

10.1 Introduction

Humans possess an exceptional ability to acquire and generalize manipulation "strategies." Even from a single demonstration of using a soup ladle to reach a distant object (Fig. 10-1a), we can generalize and reuse this strategy in various novel scenarios: to different object positions and sizes, and even to diverse object categories like forks and hammers. Furthermore, humans can combine these strategies to devise long-term plans, perhaps using a soup ladle to hook an apple, place it into a bag, and move the bag to a shelf, dramatically expanding the scope of our manipulation abilities.

A salient feature of these manipulation tactics is that they can be expressed as a sequence of *basis manipulation operations*, characterized by varying contact modes between robots and objects. For example, as illustrated in Fig. 10-1b and c, the "hook-using" strategy comprises a series of four contact modes: the free movement of the arm, tool grasping while applying contact force between the tool and the target, tool placement, and ultimately, target grasping. The continuous parameters of these operations, in principle, can be produced by generic samplers and motion planners, but planning in terms of these generic basis operations can be very slow due to a long planning horizon with substantial branching due to choice of basis operations and continuous parameters.

To tackle these challenges, this chapter presents a framework that equips machines with the ability to *learn, generalize, and reuse* such manipulation strategies, referred to as "*mechanisms*," through a single demonstration and subsequent self-play in a distribution of target problems. The key insight driving our framework is the characterization of each mechanism as a sequence of contact mode changes between the robot and objects, complemented by a specialized sampler that generates grasps, contacts, and trajectories tailored specifically for the mechanism. Our framework takes an explanation-based learning approach [Baillargeon and DeJong, 2017; Segre and DeJong, 1985], departing from conventional methods that learn policies or parameterized trajectories from large numbers of demonstrations. In particular, our framework extracts an abstract representation that



(a) Demonstration of using a soup ladle to reach for the spoon

(b) The contact mode graph between objects, with continuous parameters.

(c) A sequence of contact mode graphs in the process of using the soup ladle to reach for the spoon.

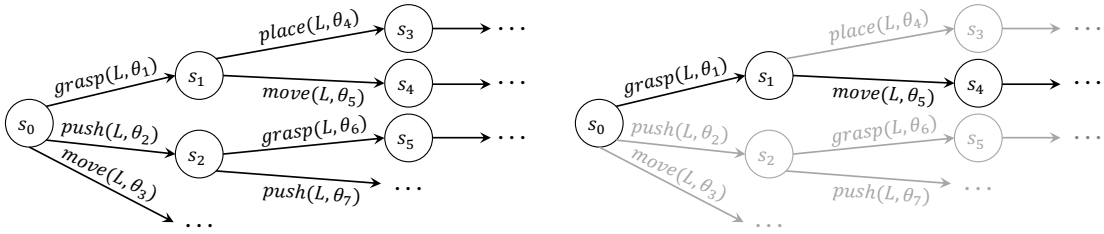
Figure 10-1: **Reusable manipulation strategies.** A single demonstration can teach a reusable multi-step manipulation strategy.

explains the underlying contact interactions between objects from the demonstration, then during the self-play stage of mechanism learning, the agent explores feasible actions that align with the demonstrated contact mode sequence, generalizing to different objects and initial configurations. Leveraging successful trials from self-play, we train samplers that are tuned specifically for each mechanism. The learned mechanisms and samplers can then be recombined to efficiently devise long-horizon plans for novel goals in novel environments, by reducing the effective search horizon and focusing the sampling process during planning.

This formulation introduces a significant capacity for generalization via abstraction and compositionality: by extracting the contact-mode sequence from the demonstration, we retain its most causally important aspects while abstracting away many irrelevant details, and by representing learned mechanisms in a form similar to basis operations, we are able to leverage general-purpose task-and-motion planners to obtain a truly compositional system.

The chapter is organized as follows. First, we present a novel representation of complex manipulation actions in terms of mechanisms, an algorithm for learning new mechanisms from a single demonstration and subsequent self-play, and a planning framework for integrating new mechanisms with other manipulation primitives, including those that are briefly dynamic [Mason, 2001], to solve novel problems.

Second, in Section 10.5, we address the limitations of this framework in generalizing to previously unseen objects. While the basic approach requires either training on datasets of object models or performing expensive random contact-point search at inference, we propose MAGIC (Manipulation Analogies for Generalizable Intelligent Contacts). MAGIC leverages a single demonstration to extract a sequence of contact waypoints, and then matches them to unseen objects through a global-to-local analogy process: pretrained visual features provide coarse correspondence across object categories, while local curvature ensures stability and physical plausibility. This approach enables one-shot generalization of manipulation



(a) Search in the large hybrid space of discrete mode families and continuous motion parameters.

(b) Search in the hybrid space guided by the mechanism. Only continuous parameters needs to be searched.

Figure 10-2: **Planning with basis manipulation operations and mechanisms.** (a) Planning with basis manipulation operations usually involves a search in the hybrid space of discrete contact mode families (free, holding, pushing, etc.) and continuous parameters (grasping pose, moving trajectories, etc.). (b) Learning mechanisms can help reduce the search space by specifying a sequence of discrete transitions between contact mode families.

strategies to novel tools and objects, significantly improving efficiency and robustness.

Third, in Section 10.6, we extend this line of work toward learning more complex interactions with objects, beyond making and breaking contacts, including operating kinematic mechanisms such as switches and drawers. We introduce KALM (Keypoint Abstraction using Large Models), a framework that prompts large vision-language models to propose candidate task-relevant keypoints, verifies their consistency across a handful of demonstrations, and distills them into compact abstractions. These keypoints are then used as part-relative action frames for training a keypoint-conditioned diffusion-based generator for contact points and robot trajectories. KALM enables robots to acquire versatile manipulation skills with minimal demonstrations, while generalizing effectively across variations in object shapes and poses.

10.2 Planning with Contacts and Mechanisms

Our framework is based on hybrid task and motion planning. We begin with a small set of generic *basis manipulation operations* corresponding to different *contact mode families* [Cheng *et al.*, 2022] between robot and objects (Table 10.1). Sequences of these basis operations form a rich class of manipulation strategies, such as using soup ladles to reach for distant objects, or pushing plates to the table edge to pick them up. In theory, a complete search algorithm could discover such strategies and find sequences of them to solve difficult novel problems, without any demonstrations at all, as illustrated in Fig. 10-2a. However, planning at this level is inefficient to the point of infeasibility. Therefore, in this chapter, we will explore learning "macros" of basis operations, using a compatible representation that will allow learned mechanisms to, themselves, be composed to solve

```
(:action move-with-contact
 :parameters (?tool - item ?target - item ?support - item
              ?param - contact-move-param ?qt - trajectory)
 :precondition (and
   (holding ?tool)
   (support ?target ?support)
   (valid-contact-move-param      ... ?param) ;; points+normal
   (valid-contact-move-trajectory ... ?param ?qt))
 :effect (and
   (assign (robot-config) ...)   ;; update robot position
   (assign (pose ?tool)   ...)   ;; update tool position
   (assign (pose ?target) ...)  ;; update target position
```

(a) The modeling of the robot basis operation *move-with-contact* using a STRIPS-like syntax..

```
(:mechanism hook
 :parameters (?tool - item ?target - item ?support - item ...)
 :precondition (and  ;; the initial contact mode graph
   (handsfree) (support ?tool ?support) (support ?target ?support))
 :certified (and     ;; the goal of the mechanism
   (support ?tool   ?support)
   (holding ?target))
 :actions (ordered   ;; the sequence of contact primitives
   (grasp     ?tool ?support)
   (move-cont ?tool ?target ?support)
   (place     ?tool ?support)
   (grasp     ?target ?support)))
```

(b) The precondition (initial contact modes), goal (target contact modes), and the sequence of actions for a mechanism.

Figure 10-3: **The syntax used to model basis operations and mechanisms.**

harder problems.

10.2.1 Basic Domain Representation

We adopt a representation similar to those used in the task and motion planning (TAMP) literature [Garrett *et al.*, 2021]. Formally, given a space $\mathcal{S}$ of world states, a TAMP problem can be defined as a tuple $\langle \mathcal{S}, s_0, \mathcal{G}, \mathcal{A}, \mathcal{T} \rangle$. Here, $s_0 \in \mathcal{S}$ denotes the initial state, which includes the shapes and poses of all objects. $\mathcal{G} \subseteq \mathcal{S}$ represents the goal specification, often expressed as a logical expression evaluated based on a state (e.g., *holding*(Spoon)). $\mathcal{A}$ denotes a set of continuously parameterized actions that the agent can execute, such as grasping and placing objects. Finally, $\mathcal{T}$ is a partial environmental transition model defined as $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. Each action a is parameterized by two functions: the precondition function pre_a and the effect function eff_a. The semantics of these functions is that for any state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$, if $pre_a(s)$ holds, then $\mathcal{T}(s, a) = eff_a(s)$.

State representation. An environmental state is represented as a tuple $s = (\mathcal{U}_s, \mathcal{P}_s)$. $\mathcal{U}_s$ denotes a set of objects, including the robot and other physical objects, and we assume it is fixed during the execution of actions. Objects in $\mathcal{U}_s$ will be referred to using names such as SoupLadle and Floor. The set $\mathcal{P}_s$ contains *state variables* (atoms in STRIPS). Each state variable contains a predicate name (e.g., *pose*), a list of object arguments (e.g., SoupLadle), and a value (e.g., the pose of the soup ladle in $SE(3)$). In addition to shapes and pose variables, $\mathcal{P}_s$ also contains two sets of variables to represent the contact mode graph: *holding*(*?x*) and *support*(*?x*, *?y*), where *?x* and *?y* are instantiated with all objects in $\mathcal{U}_s$. These variables describe whether the robot is holding an object *?x* and whether the object *?x* is supported by *?y*, respectively*.

*Following the STRIPS convention, we will be using names such as *?x* and *?y* for variables and strings such as SoupLadle for objects.

Name	Args.	Description
transit	⟨*empty*⟩	Move the robot without object collisions.
transit-cont	*?x, ?s*	Move without holding anything, but have a contact with *?x*. *?x* is supported by *?s*.
grasp	*?x, ?s*	Grasp the object *?x* that is currently supported by *?s*.
place	*?x, ?s*	Place object *?x* onto *?s*. *?x* should be currently held by the robot.
move	*?x*	Move while holding *?x*.
move-cont	*?x, ?y, ?s*	Move while holding *?x* and *?x* have contact with *?y*. *?y* is supported by *?s*.

Table 10.1: **List of basis operations.** Continuous parameters are omitted. "cont" refers to "contact."

Basis operators. The basis operators are specified as parameterized *operator schemas*, ⟨*name*, *args*, *precond*, *effect*, *sampler*⟩, where *name* is the name of the schema, *args* is a list of arguments, including both object arguments and continuous parameters. These continuous values can be generated by invoking the *sampler*, possibly conditioned on other aspects of the state, such as the shapes of the objects involved. The precondition *precond* and effect *effect* are logical expressions over variables in *args* and will be evaluated at the current state. A schema can be *grounded* into a concrete basis operation *a* by specifying its arguments. See Figure Fig. 10-3a for a concrete example and Appendix J.1.3 for more discussion.

10.2.2 Mechanisms

A manipulation *mechanism* is defined as a sequence of basis operations with a specialized sampler, in order to accelerate planning with generic basis operations. Formally, each mechanism is represented as a tuple of ⟨*args*, *precond*, *certified*, *actions*, *sampler*⟩. *args* is a set of arguments. *precond* is the initial contact mode graph including *holding* and *support* relations. *certified* is the goal of the mechanism, which usually specifies the final contact mode graph. *actions* is an ordered list of primitive operations. *sampler* is the specialized sampler that can generate continuous parameters for all basis operations in *actions*. Fig. 10-3b illustrates the definition of the "hook-use" mechanism. In this case, three objects are involved: the tool object (e.g., the soup ladle), the target object (e.g., the spoon), and the support object (e.g., the floor). The goal of the mechanism is to grasp the target object that was initially out of reach. This macro contains a sequence of four basis operations: grasping the tool from the support, moving while holding the tool and "indirectly" pushing the target object, placing the tool back to the support, and finally grasping the target object. It also has an associated sampler that generates feasible grasps of the tool and contacts between the tool and the target object.

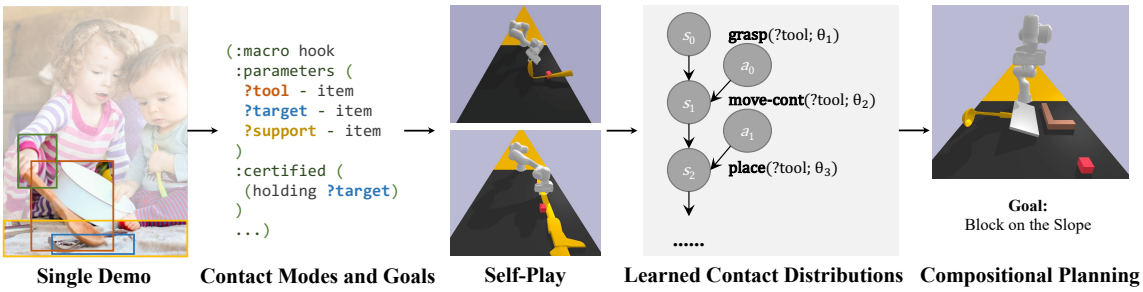


Figure 10-4: **Learning of manipulation strategies.** Learning process: extract contact modes and goals from demonstration; train specialized sampler via trial-and-error in new situations; add new mechanism to planner.

10.2.3 Planning with Basis Operators and Mechanisms

To plan using the basis operations, mechanisms, and samplers, we employ a simple bilevel search approach, similar to the adaptive algorithm in PDDLStream [Garrett *et al.*, 2020a]. Illustrated in Algorithm 13 in Appendix J.1.1, we first apply symbolic STRIPS search algorithms with the fast-forward heuristic [Hoffmann and Nebel, 2001] to explore discrete action plans (i.e., candidate sequences of basis operations and objects to move). Subsequently, we use samplers to find suitable continuous parameters. We iteratively call the samplers associated with each operation to generate continuous parameters, such as grasps, contact surfaces, and trajectories. We simulate the grounded operators to verify their effects, and backtrack to a new discrete plan if continuous parameters cannot be found. Mechanisms can be directly integrated into the bi-level search-based planner as additional operators, with their own specialized samplers. Intuitively, as illustrated in Fig. 10-2b, incorporating mechanisms in search prunes out the branching factor of possible contact modes and also improve efficiency in sampling continuous parameters.

The use of a physical simulator to verify action effects enables us to handle *briefly-dynamic* tasks, in which the robot controllers are position-based, but may initiate situations in which the objects experience acceleration and velocity before they reach a stable configuration. Examples include objects sliding down inclines, or tipping upward when weights are placed on them (see Appendix J.1.2).

10.3 Learning New Mechanisms

Our goal is to learn a new *mechanism* from a single demonstration and a distribution of target problems. The demonstration includes a sequence of robot actions and object contacts, as well as a human-specified goal, which will be the *certified* effect in the new mechanism definition. We assume access to an environment simulator that can generate random initial

configurations of objects such that the target mechanism is applicable. For example, in the *hook-use* case, the environment contains two objects of various categories placed on the table so that one object is within reach and the other object is out of reach. The overall framework is depicted in Fig. 10-4: The learning algorithm first extracts the contact-mode changes in the demonstration, resulting in a sequence of basis operations. Next, it generates self-play trajectories that align with the basis-operation sequence but now with novel objects in novel initial configurations (e.g., using spoons to reach for forks). This self-play step involves trial-and-error interaction with the environment. Based on the successful trajectories, we learn a sampler that generates mechanism-specific contacts between objects (e.g., grasp of tools in order to reach for distant objects), and finally add it to our repertoire of planning operators.

10.3.1 Extraction of Preconditions and Operation Sequence

We first segment the demonstration trajectory based on robot-object contacts: free motion, prehensile "holding" of an object" and non-prehensile "pushing" of an object. Each segment will correspond to one step in the mechanism macro. Next, for each segment, we build a contact graph between the robot hand, the object that is in contact with the robot (via holding or pushing, if any), and the object that is in contact with the held object ("indirect contact"). Finally, we add all the objects that support the objects (exerting forces that point in the $+z$ direction). By segmenting the raw sequence at the points where the contact modes change, we obtain the sequence of basis operations. The precondition of the mechanism operator corresponds to the initial contact modes. After extracting the initial contact modes and the basis operation sequence that are grounded on concrete objects in the demonstration (the soup ladle, the spoon, and the floor), we *lift* them into an abstract mechanism definition by replacing concrete object names with variables, as illustrated in Fig. 10-3b.

10.3.2 Sampler Learning

The sampler learning process is an iterative exploration within a simulated environment. Algorithm 7 describes the high-level process of learning a sampler based on the success of generated samples in achieving the goal. In each iteration, we randomly sample an initial configuration from the simulator and attempt to execute the mechanism on objects present in the environment. Notably, instead of searching for a plan to accomplish the mechanism's goal with all available basis operators, we require the search algorithm to adhere to the sequence of basis operations derived from the demonstration. Specifically, we fix the discrete-level plan and employ generic samplers for grasping, contact, and trajectory generation to locate plans that satisfy the goal. All continuous parameters sampled during

Algorithm 7 Sampler Learning Algorithm

procedure SAMPLERLEARNING(env, m)
  Initialize classifiers f_i and dataset $\mathcal{D}_i \leftarrow \emptyset$ for each continuous parameters i in m.
  for each iteration **do**
    $s_0 \sim env.reset()$ ▷ Sample an initial configuration
    $gm \leftarrow$ RandomGrounding(m) ▷ Apply the mechanism on a random set of objects
    $plan \leftarrow$ CONTINUOUSSEARCH($s_0, gm.certified, gm.actions$)
    for each continuous parameter i **do**
      for all θ sampled in CONTINUOUSSEARCH for parameter i **do**
        if θ is in $plan$ **then** Add $(\theta, 1)$ to $\mathcal{D}_i$ ▷ Successful samples
        else Add $(\theta, 0)$ to $\mathcal{D}_i$ ▷ Failed samples
      Update classifier f_i using L and $\mathcal{D}_i$

the search will be labeled as "1" or "0" based on whether they present in the successful plan. Note that even if the environment does not contain distractors and the mechanism is always applicable, we still need to explore different choices of arguments (e.g., use which object to reach for the other one).

Leveraging this labeled dataset of samples, we can train an additional sampler that generates mechanism-specific samples of continuous parameters. For example, in a scenario where a soup ladle is used to reach distant objects, it is favorable to grasp the handle—a more mechanism-specific action, rather than generic grasps. This can be formalized as learning a distribution of "successful" continuous parameters for actions within a mechanism. We do this by training a score function to rank samples produced by the generic samplers[†]. In essence, for each continuous parameter in the mechanism (grasping poses, contact surfaces, and trajectories), given a dataset of samples and their success labels $\mathcal{D} = (\theta, label)$, we train a classifier f that estimates a scalar value in $[0, 1]$ representing the probability that θ can result in a successful application of the mechanism, using binary cross-entropy loss: During deployment, we use the generic sampler to generate a batch of samples, which we sort based on the predicted likelihood of success.

Note that although we train classifiers for individual parameters, these classifiers are designed to be conditioned on previously generated parameters as defined in the basis operation schema. Hence, their sequential application is capable of generating a joint distribution of successful parameters for mechanism applications in an "auto-regressive" manner. The benefit of this factorized approach, as opposed to learning a single classifier over the concatenation of all parameters, is its efficiency during the search process. If the execution of one step in the mechanism fails, the search algorithm can immediately backtrack, thereby averting wasteful continuation into subsequent steps.

[†]More sophisticated generative models will work, especially when the samples are hard to generate.

10.3.3 Implementation

Our algorithm for learning mechanisms is general purpose. In order to demonstrate its value, we have implemented a concrete system with the basis operators in Table 10.1.

Samplers for Basis Operators There are three types of continuous variables to be sampled for the basis operators described in Table 10.1: grasping poses relative to an object (represented as $SE(3)$ poses of the end-effector), placement poses (represented as $SE(3)$ poses in the support object), contacts between two objects (represented as the $SE(3)$ pose of object 1 in the frame of object 2), and robot arm trajectories (represented as a sequence of joint configurations). The samplers are designed to be generic, relying solely on geometry and not specific object semantics (e.g., soup ladle grasping). We include one here as an example and describe the full list in Appendix J.1.4.

Object Contact (C). For both robot-object and object-object contact, the object contact sampler, $C(O_1, O_2, T_{o2}, O_s, T_s)$ takes five arguments, including the current holding object O_1 (or the robot gripper itself when not holding anything), the object to contact O_2 and its current pose T_{o2}, and the object that supports O_2: O_s and its pose T_s. It first randomly samples two surfaces, represented as a pair of (point, unit normal), (p_1, n_1) and (p_2, n_2) on O_1 and O_2 respectively. Since we do not consider pushing O_2 "towards" the supporting object O_s, we additionally require that n_2 is perpendicular to n_s, which is the direction of the support force from O_s to O_2. Next, it solves for a transform T on O_1 such that $Tp_1 = T_{o2}p_2$ and $Tn_1 = -T_{o2}n_2$ (essentially place p_1 on O_1 at p_2 and pointing towards n_2 to excert force).

Learned Samplers Each mechanism has specialized samplers for all of its continuous parameter, organized into a DAG, in which each parameter is sampled conditioned on its parents in the graph (a.k.a. previously sampled parameters). Each sampler has a scoring classifier that processes the target parameter along with any values it is conditioned on and predicts a success likelihood in $[0, 1]$. We employ various encoders for different parameter types. For object shapes, we use a PointNet++ encoder [Qi *et al.*, 2017] to process the point clouds. For poses, including a 3D translation and a 4D quaternion, we utilize Multi-Layer Perceptrons (MLPs). For contact information, we encode the contact points and normals using MLPs as well. Each of these encoders processes their corresponding inputs into a fixed-length embedding, and the embeddings are then concatenated into a single vector. Finally, we apply a linear transformation followed by a sigmoid activation to output the classification result. We train each classifier on 100 samples. For trajectory-typed parameters, we only apply classifiers to object-object contact trajectories encoded as the contact normal direction and the distance. We do not consider encoding and classifying the actual robot arm trajectory for grasping and placement motions, although it is, in principle,

Method	Edge	Hook	Lever	Poking	CoM	Slope&Blocker
Basis Ops Only	89.45±5.53	>600	523.18±9.22	>600	19.30±2.82	>600
Ours (Macro)	8.34±2.57	30.82±5.78	**1.38**±0.31	494.30±50.01	17.58±1.27	148.57±10.30
Ours (Macro+Sampler)	**0.57**±0.05	**3.84**±1.56	1.55±0.29	**97.76**±10.67	**0.97**±0.09	**4.11**±0.94

Table 10.2: **Planning time comparison for manipulation strategies.** Average solution time and stderr of 10 trials, All methods have a 10 mins timeout.

possible to encode them using appropriate encoders.

10.4 Experiments

We conducted experiments in the PyBullet simulation environment, to evaluate the ability of our system to learn individual mechanisms and its ability to reuse learned mechanisms in novel tasks. We describe real-world deployment of the system in Appendix J.2.3 and provide videos on our website.

10.4.1 Learning Mechanisms from Single Demonstrations

Setup. Our evaluation encompasses six distinct mechanisms, grouped into two categories: the first four tasks assess tool-use, including ***Edge***: pushing objects to the edge of a table for pickup, ***Hook***: using tools to reach for distant objects, ***Lever***: flipping objects using heavy objects as levers, and ***Poking***: using tools to poke objects out of a tunnel. The remaining two tasks fall under the "reasoning about stability" category, including ***Center-of-Mass***: achieving stable object placement on another object, and ***Slope-and-Blocker***: using objects as blockers to prevent objects from falling off inclined surfaces (illustrated in Fig. 10-5). The object models are blocks, bricks, bowls, plates, documents, spoons, forks, soup ladles, hairbrushes, hammers, and calipers. The demonstrations are created by executing a manually written script in one specific initial configuration.

During training and testing, each method has access to a distribution of initial configurations and goals. Each task consists of a randomly sampled initial configuration that includes target objects placed on the table and a specific goal to be achieved (e.g., holding one of the target objects). We design the distribution of initial object placements to ensure the feasibility of the corresponding mechanism. For example, in the "hook-use" mechanism, we randomly place two objects on the table. Object 1, which is within the robot's reach, is selected from the following categories: soup ladle, hammer, spoon, and hairbrush. Object 2 is a box that is initially placed outside of the robot's reach. We provide more environmental details in Appendix J.2.1.

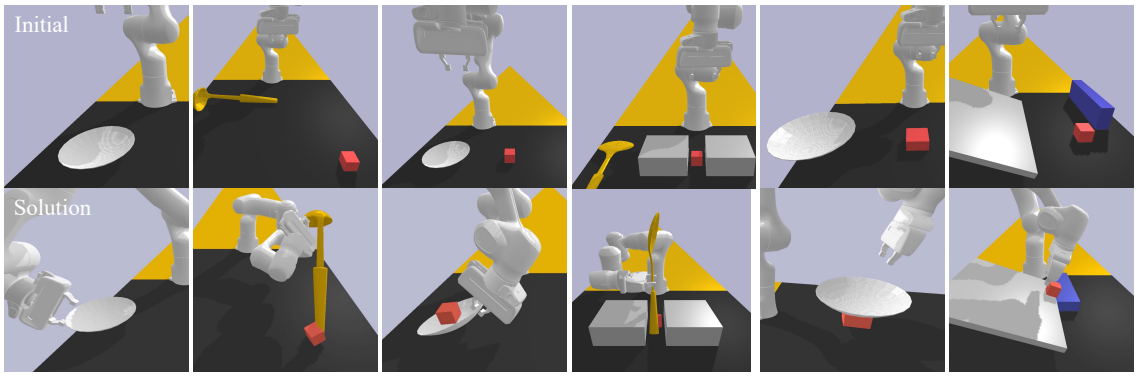
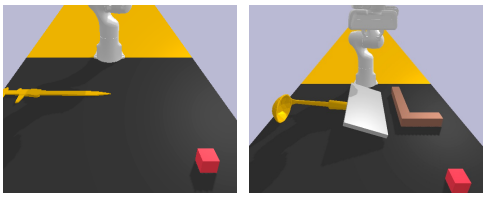


Figure 10-5: **Illustration of six mechanisms learned by our algorithm.** Top row: initial configuration. Bottom row: solution found by our sampling-based planner. P: Plate. B: Block. S: Slope.

Results. Table 10.2 summarizes the results of our experiments comparing our method to planning with the basis operators only, as well as to an ablation of our method in which we learn the basis operation sequence for the mechanism but do not learn specialized samplers. Due to the inclusion of novel object instances and categories in the test environments, simpler baselines, such as replay of the demonstration trajectory, have zero success rate, and are not included. The table shows the average time needed for each planning algorithm. Our method effectively learns mechanisms for all tasks. The macro-only version occasionally timed out, emphasizing the importance of both sequence and sampler learning. As an example, consider the Lever task, in which the robot needs to position an object on a plate to flip the plate up. Having prior knowledge of this strategy at the abstract basis operation level significantly simplifies the task. The learning of a mechanism-specific sampler, in this case, does not offer much additional benefit. On the contrary, task CoM presents a different scenario: here, the robot is tasked with learning a distribution for stably placing an object on a small block, which benefits greatly from learning a specialized sampler. The poking task is difficult, even for learned samplers because it involves a arm motion planning problem with a very small collision-free region. We discuss more details on the importance of sampler learning in Appendix J.2.2.

10.4.2 Planning with Learned Mechanisms

Setup. To test the compositionality of mechanisms, we evaluate different algorithms on two novel complex tasks, illustrated in Fig. 10-6. In **Push-then-Pick-then-Hook**, the agent needs to utilize a thin caliper, placed on the table to reach for a distant block. However, the caliper must first be pushed to the table side to enable a successful grasp. In **Hook-**



Task 1: hold(Block) **Task 2:** support(Block, Slope)

Figure 10-6: Illustration of compositional tasks.

Method	Task 1	Task 2
Basis Ops Only	0%	0%
Ours (Macro)	20%	0%
Ours (Macro + Sampler)	100%	90%

Table 10.3: Average success rate on novel compositional tasks in 10 runs. Timeout is 10 minutes.

then-Place-on-Slope, the agent needs to use a soup ladle as a tool to reach for a distant object. Subsequently, the agent must use either the soup ladle or a brown brick to act as a blocker, preventing the object from falling off a slope. There is no additional training—we used mechanisms learned during the previous experiment. We design the test distribution of object placements to ensure tasks being feasible.

Results. Table 10.3 presents the planning time for all the evaluated methods. In scenarios with numerous objects available for interaction, searching directly for low-level manipulation primitives without the guidance of useful mechanisms can be extremely slow. However, using mechanisms as "macros" in the search process significantly enhances efficiency, enabling the system to solve a greater number of tasks within the given time limit. The learned samplers further improve the overall efficiency.

10.5 One-Shot Manipulation Strategy Learning

Next, we address the limitations of mechanisms in generalizing to previously unseen objects. In this section, we develop MAGIC (Manipulation Analogies for Generalizable Intelligent Contacts), a framework for one-shot strategy transfer. MAGIC extracts contact waypoints from demonstrations and establishes functional correspondences on novel objects through a global-to-local analogy process: coarse shape alignment using pretrained visual features followed by fine-grained geometric matching based on local curvature. By reasoning directly about affordances rather than memorizing object instances, MAGIC enables efficient generalization to diverse tools and object categories. This approach enables one-shot generalization of manipulation strategies to novel tools and objects, significantly improving efficiency and robustness.

Shown in Fig. 10-7, given a *single* reference action trajectory (e.g., using a hook to reach for a distant object) and a novel scenario (e.g., with different tools and different objects), the goal of the algorithm is to generate a sequence of robot actions that apply a "similar" strategy to the test objects specified by users: in this example, having the target object moving along

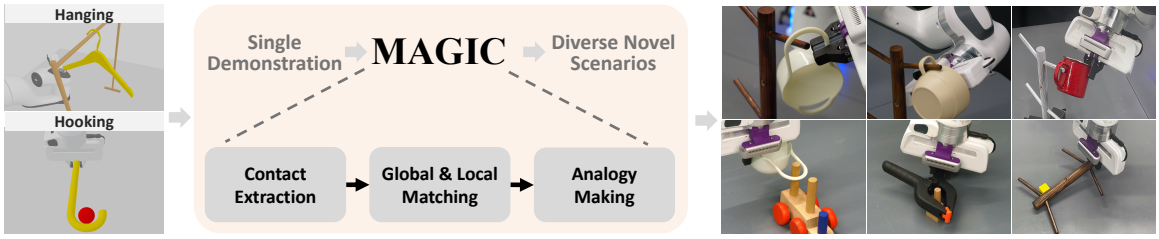


Figure 10-7: **Overview of MAGIC.** MAGIC (Manipulation Analogies for Generalizable Intelligent Contacts), a pipeline that is capable of learning manipulation strategies from single demonstrations and applying them to novel objects.

a certain direction for a given distance. MAGIC extends two critical insights into a broad class of manipulation strategies. First, many strategies such as hooking, hanging, hammering, pushing, reaching [Qin *et al.*, 2020; Turpin *et al.*, 2021], stacking, pouring [Lai *et al.*, 2021; Sieb *et al.*, 2020; Xu *et al.*, 2021b], and cutting [Xu *et al.*, 2021b] can be characterized by a sequence of contact waypoints (i.e., the order in which contacts between objects and robot bodies are made); second, these contacts are characterized by forceful affordances between object pairs: a specific pair of contact points on two objects would enable the application of forces along certain directions. However, searching for contact points that would enable the specific affordance is generally challenging due to complex constraints on reachability, collision avoidance, and motion stability.

MAGIC tackles these challenges by combining data-driven and analytic approaches to generate contact waypoints in novel scenarios. In particular, it first extracts the sequence of contacts among objects in the reference trajectory and then proposes (pairs of) contact points that have similar global and local shape properties as the contact points in the reference, which can be used as guidance for motion planning or motion retargeting. Finally, it leverages a physical simulator to select the contact and the action sequence that achieves the goal. Our key innovation lies in a novel global-to-local matching algorithm to find functional correspondences between the target objects and reference objects. Intuitively, a "good" contact point would satisfy both a global and a local matching property. First, the points on two objects should be on similar parts of the global shape (e.g., in the hook-using example, we need a contact point on the tool that is at the end of a long rod). Second, the hooking contact point should have a matched local curvature with the target object being hooked so that we can execute the actions stably. Therefore, we propose to use a pretrained visual feature-based correspondence matching to resolve the global matching property. This enables us to quickly search over different parts of the objects but the resulting contact point is usually not precise. Next, we use a local curvature-based matching algorithm to find the best contact point within a local region of the previously proposed contact point, which gives us precise and physically plausible (e.g., collision-free and physically stable) contacts.

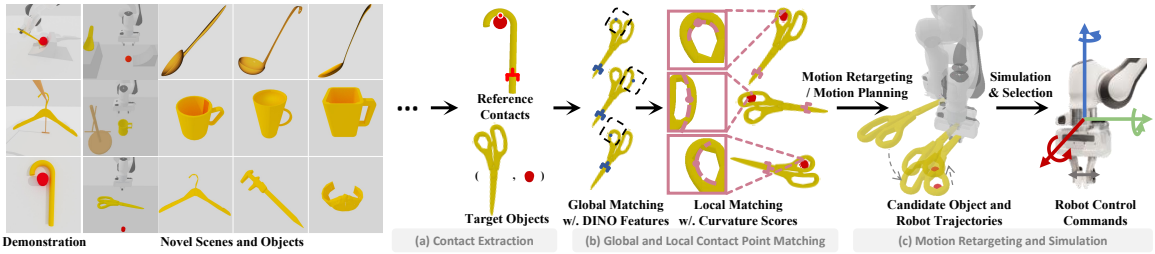


Figure 10-8: **The overall pipeline of MAGIC.** (a) We first extract contact points from the reference trajectory. (b) Then, we compute a global and local contact point matching score to select candidate contact points on novel objects. (c) The generated contact points will be used for motion retargeting or motion planning, and the final motion will be simulated and verified by a physical simulator.

Overall, MAGIC tackles the problem of one-shot manipulation strategy learning by making analogies in contact waypoints. We validate the effectiveness of our approach on three challenging tasks: scooping a ball against a concave arc with a spoon, hanging a mug onto a mug tree, and using tools to hook objects of varying sizes. Compared to global shape-matching algorithms, our framework achieves significant improvements when the reference objects are from different categories than the test objects. Compared to local shape-matching and simulation-based approaches, our framework is orders of magnitude faster — for most test objects, we need to run simulations on fewer than three candidate contact points to find a solution. Finally, compared to pretrained feature-matching-based approaches, our method finds more precise and physically plausible solutions. We propose MAGIC (manipulation analogies for generalizable intelligent contacts), a novel approach for fast and generalizable manipulation strategy learning from a single demonstration. Fig. 10-8 shows the overall framework. First, we extract the contact points among objects from the reference trajectory. Subsequently, we find the candidate contact points that support the functional affordances on unseen test objects through a global-to-local matching process, in which we first perform coarse correspondence point matching using pretrained vision transformer (ViT) features DINOv2 [Oquab *et al.*, 2024] (Section 10.5.2), followed by a local alignment based on shape curvatures to find stable local contact patches (Section 10.5.3). Finally, the candidate contact points and their matching scores can be used for tool selection, for retargeting reference object trajectories, or as waypoints for motion planning (Section 10.5.4). The generated trajectories will be verified in a physical simulation and then output to the robot for execution.

10.5.1 Problem Definition

We introduce the task of one-shot manipulation strategy learning. Specifically, we start with a single demonstration involving the SE(3) trajectories of objects that execute a particular manipulation strategy to achieve a goal (e.g., hanging a hanger) in the reference scene. Our goal is to generate the trajectories of objects in a target scene that apply a similar manipulation strategy to a different object (e.g., attaching a mug to a mug tree). We simplify this problem by assuming the robot is only manipulating one of the objects (i.e., the "tool" that the robot is holding), and the desired motion can be explained by achieving a sequence of contact waypoints. Many rigid-body tool-using tasks such as hooking, poking, stacking, and funneling, all fall into this category. For now, we assume that the strategy to apply and the manipulanda in both the reference and target scenes (e.g., the hanger or the mug) are already identified, and later we will discuss strategies for selecting the best tool object to accomplish a certain task. Therefore, the problem can be cast as making the analogy between the reference object motion and the target object motion, taking into consideration other environmental constraints such as robot reachability and collisions.

Based on our insights into decomposing manipulation trajectories into contact point sequences, this task further reduces to establishing a functional correspondence between the demonstrated object and the novel object, which can be represented as an SE(3) transformation between two rigid bodies for each contact waypoint. For tasks considered in this section, we only consider scenarios where there is a single contact waypoint responsible for the target motion (but it generalizes to finding correspondences in multiple waypoints), and there is a reduced degree of freedom that can be reasoned about in 2D. In particular, we only model contacts on a canonical 2D cross-sectional view of the objects. For example, when we consider hooking objects on a table, we only consider object motions and shape properties on the surface that is perpendicular to the table's surface normal. This is equivalent to assuming access to a canonical pose of the object: for hooks, this will be the top-down view when the hook is placed on the table; for mugs, this will be the side view that contains the handle of the mug. In this section, we assume that this canonical view is given, and in general, it can be predicted by external modules. Therefore, now, our goal is to establish a single correspondence between two 2D images of the objects in E(2) (translations, rotations, and reflections in 2D) plus scaling; and then we can recover the SE(3) correspondences based on the canonical object poses.

Overall, the input to MAGIC is an image I_T of the reference object T, an image $I_{T'}$ of the target object, and a point of interest p_T on the reference image. Our goal is to find the corresponding point $p_{T'}$ on $I_{T'}$. When we have two objects interacting: the tool object T being directly held by the robot (e.g., the hook) and another object O in contact with T (e.g., the target object we want to hook), our goal would be to find a contact point pair $p_{T'}$

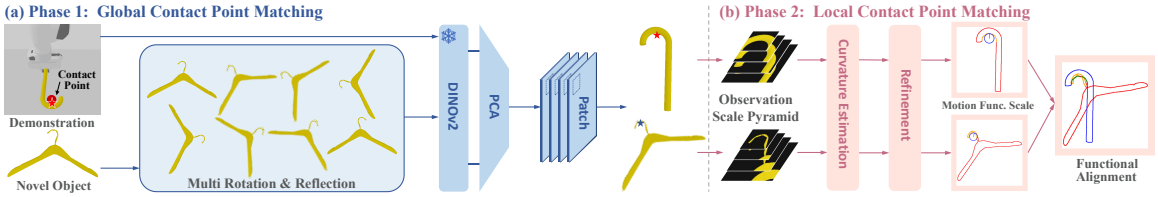


Figure 10-9: **Global and local contact point matching.** The contact point matching process consists of two phases: (a) global matching using DINOv2 [Oquab *et al.*, 2024] features; (b) local matching involving multi-scale curvature estimation and refinements using irrelevant point suppression and convexity matching.

on $I_{T'}$ and $p_{O'}$ on $I_{O'}$ given the reference $\langle T, O, p_T, p_O \rangle$ that maximizes a score function: $score\,(X, X')$, where $X = \langle T, O, p_T, p_O \rangle$ is the reference contact and $X' = \langle T', O', p_{T'}, p_{O'} \rangle$ is the target contact. In this section, we employ a two-stage global-to-local matching process, therefore, the *score* function will be composed of two parts: a global matching score s_{dino}, and a local matching score s_{curv}:

$$score\,(X, X') = s_{dino}(T, T', p_T, p_{T'}) + \lambda \cdot s_{curv}(X, X'),$$

where λ is a constant hyperparameter. p_O and p_T are automatically extracted using the contact information in simulation and will be manually annotated for real-world objects (they can also be automatically recovered using a video-based contact point detector such as [Ehsani *et al.*, 2020]). After generating top candidates for $p_{T'}$ and $p_{O'}$ based on *score*, for additional validation, we will simulate the computed trajectory with the target object and select the first one that succeeded in the task in simulation. All tasks studied in this section involve only stable quasi-static motion. Therefore, they can be simulated using mild assumptions on point clouds, uniform object densities, and frictions. When the simulation is unavailable in challenging dynamic tasks (e.g., hammering), we fall back to using the highest-scoring contact match.

10.5.2 Global Contact Point Matching with Pretrained Features

The global matching score s_{dino} is only computed between the tool objects T and T'. In this stage, we will find a candidate set of contact points "globally" on the target object T', leveraging visual features pretrained on large image datasets for capturing global and semantic correspondences [Zhang *et al.*, 2023]. Specifically, we adopt DINOv2 [Zhang *et al.*, 2023] as our visual feature extractor. Fig. 10-9a shows the pipeline: we first extract visual features for both objects (with different image rotations and reflections) and then find a candidate set of matching points.

Feature extraction. We feed I_T and $I_{T'}$ into DINOv2 respectively, and get feature maps F_T and $F_{T'}$. In practice, to eliminate the effects of rotation and reflection in 2D, we apply horizontal flips and 12 rotations on $I_{T'}$ and get a total of 24 images $\{F^i_{T'}\}_{i=1}^{24}$. Next, we apply a principal component analysis (PCA) on the feature vectors. Given feature maps $F_T, F^i_{T'} \in \mathbb{R}^{n\times n\times d}$, let $\mathit{flatten}(F)$ be flattened version of F in $\mathbb{R}^{n^2\times d}$, we have $W = \text{PCA}_{d,d'}\{\mathit{flatten}(F_T), \{\mathit{flatten}(F^i_{T'})\}_{i=1}^{24}\}$, where d and d' are the original feature dimension and the feature dimension after reduction, respectively, and $W \in \mathbb{R}^{d\times d'}$ is the matrix of principal component vectors. The output is $\widetilde{F} = F \times W \in \mathbb{R}^{n\times n\times d'}$.

Matching by patches. Next, we aggregate the features within a local region (instead of a single point) so as to increase the receptive field. Formally, let $\mathit{patch}(p)$ denote the $m \times m$ local area centered at point p, we find a point across all rotations of the target image $p_{T'}$ that maximizes the patch-aggregated cosine similarity with the reference point p_T:

$$s_{dino}\,(T, T', p_T, p_{T'}) = \sum_{\substack{p\in patch(p_{T'}),\\ p'\in patch(p_T)}} \left\langle \widetilde{F}^i_{T'}(p), \widetilde{F}_T(p') \right\rangle .$$

We select the top $k = 3$ matches with the highest s_{dino} to perform the local contact point matching in the next stage.

10.5.3 Local Contact Point Matching with Curvature Estimation

Although DINOv2 can propose contact points within a coarse global region, it is not accurate enough in terms of local geometric properties to support collision-free and stable physical motion. We leverage curvatures to refine the correspondence matching. Illustrated in Fig. 10-9b, our local alignment has four steps. We first construct a multiple observation scale pyramid: we estimate the curvatures of the reference contact point and the contact point proposed by DINOv2 on the target object at multiple scales. Then, based on the normal direction and the sign of the estimated curvatures, we perform irrelevant point suppression and convexity matching to find a physically plausible contact point for each scale. After that, we repeat the curvature estimation process on the updated contact points to get a more accurate curvature and normal direction. Finally, we accept the best contact point across scales.

Curvature estimation at a given scale. Given an image of the object with segmentation, we first use the Canny edge detector [Canny, 1986] to get the object edges. We define the *observation scale* as the radius of the region centered at the point of interest (e.g., the contact point) on the object edge. For a certain observation scale s of the point $\mathbf{c}$, we can use the

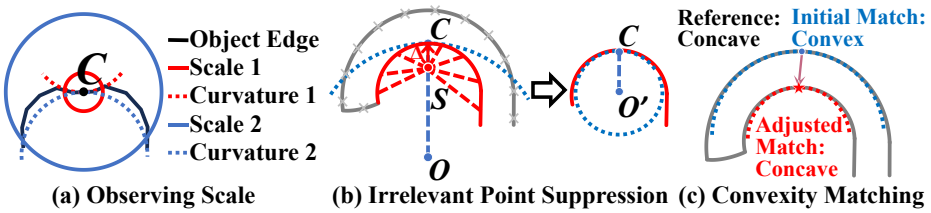


Figure 10-10: **Observation scale for curvature computation. (a)** The observing scale at which we perform estimation has an effect on both the magnitude and the sign of he estimated curvature. **(b)** Filtering out the irrelevant points can give a more accurate curvature estimation. **(c)** We perform local search within the observation region to find the contact point with the correct convexity.

edge points $\{(x_i, y_i)\}_{i=1}^n$ inside s to estimate the magnitude κ of the curvature and the radius of curvature r. Specifically, we first find the direction $\mathbf{x}'$ with the largest variance on $\{(x_i, y_i)\}_{i=1}^n$, and construct a local coordinate system $\mathbf{x}'\mathbf{y}'$ centered at $\mathbf{c}$, then represent $\{(x_i, y_i)\}_{i=1}^n$ in $\mathbf{x}'\mathbf{y}'$ to get $\{(x_i', y_i')\}_{i=1}^n$. Afterwards, we fit a parabola $y' = ax'^2$ on points $\{(x_i', y_i')\}_{i=1}^n$, i.e.,

$$a = \arg\min_a \sum_{i=1}^{n} \left(y_i' - a x_i'^2\right)^2,$$

then by the definition of curvature, we have $\kappa = 2|a|, r = 1/\kappa$..

Now, for a pyramid of observation scales $\{s_j\}_{j=1}^m$, we can compute a series of radii of curvature $\{r_j\}_{j=1}^m$. The *motion functional scale* is defined as s_j, where $j = \arg\min_{j'} \left|\frac{s_{j'}}{r_{j'}} - \alpha\right|$, where α is a parameter, and in practice, we select $\alpha = 3.5$ for all objects across all experiments. This constant corresponds to a scenario where the observation scale is similar to the radius of curvature estimated using the points inside the observation scale. Furthermore, the sign of the curvature (that is, whether the point on the curve is convex or concave) can be computed based on the mask of the object. Given both the tool object T' and target object O', we define the local matching score as:

$$s_{curv}\left(\langle T, O, p_T, p_O\rangle, \langle T', O', p_{T'}, p_{O'}\rangle\right) = \left|\frac{r(p_T)}{r(p_O)} - \frac{r(p_{T'})}{r(p_{O'})}\right|.$$

Irrelevant point suppression and convexity matching. Direct application of the curvature estimation algorithm is not robust enough for two reasons. First, the choice of the observation scale will significantly affect the estimation of the curvature. As illustrated in Fig. 10-10a, selecting a very small scale (scale 1) will make the estimation sensitive to local noise. Second, for large observation scales, irrelevant points (i.e., points on a different "edge" of the object) will be included and inject noise in the estimation, especially for thin objects as shown in Fig. 10-10b.

To eliminate these irrelevant points around the contact point of interest, we compute the curvature twice. First, we include all points within the observation scale. After computing the initial contact point C on the edge of the object and its curvature, we pick a point S that is at a small distance Δ from the contact point C along the direction of the radius of curvature $\overrightarrow{CO}$ in the initial estimation. Then, we select all points on the same "edge" as C on the side of S. This is done by emitting "rays" in all directions from S and for each ray selecting the point that is hit first. We then use these points to estimate the curvature again, as shown on the right of Figure 10-10b. Moreover, we ensure that the curvature sign (i.e., the convexity) of the corresponding point in the target image is consistent with that in the reference image by performing a local search within the observation region of the target image, which is depicted in Fig. 10-10c.

10.5.4 Generating Object Motions by Making Contact Analogies

So far we have presented a general mechanism for finding geometric functional correspondence between reference and target objects, and it can be used in many downstream modules. In the one-shot manipulation strategy learning setting, we consider three particular use cases.

Object motion retargeting. This is the most straightforward way to generate target object motions assuming there is a single contact that accounts for the motion. Given the reference contact point p_T, the direction vector $\mathbf{v}$ of the view angle, the normal vector $\mathbf{n}_T$ of p_T estimated from curvature calculation, we can construct two local frames for the reference and the target object, and directly retarget the reference object motion by aligning two local frames.

Motion planning based on contact waypoints and force directions. We can generate object motion by leveraging analogical contact points as waypoints for a motion planner. For example, if our goal is to hang an object at a particular position, we can first compute the hanging pose by making analogies with the reference object pose and use a collision-free motion planner to generate the robot motion. This also applies to hook-using tasks where the goal is specified as to apply a particular force along a target direction on an object.

Tool selection. Our pipeline also supports tool selection, where the goal is to select one "tool" object (e.g., a hook) from a set of available objects that can best execute the strategy in the new scenario, given the other object to manipulate (e.g., the object to be hooked). We apply our algorithm on all available tools, and select the tool with the highest score.

10.5.5 Experiment Setup

In this section, we have conducted experiments both in simulation and in the real world. For simulation experiments, we adopt the simulation environment SAPIEN 2 [Xiang *et al.*, 2020] using a Franka Emika Panda arm. We evaluate different one-shot manipulation strategy learning algorithms on three tasks, as shown in Fig. 10-8: in ***Scooping***, the robot need to scoop balls of different sizes with various spoons against a concave arc, given a demonstration with a reference spoon; in ***Hanging***, the robot need to hang a mug onto a mug tree, given a demonstration of hanging a hanger on a rod; in ***Hooking***, the robot selects a tool from a set to hook objects of varying shapes and sizes, given a demonstration of hooking a ball with a hook. For all tasks, we have two variants: *Floating-Gripper (FG)* where the tool will be manipulated by a floating gripper, and the harder *Arm* variant where the tool needs to be grasped and manipulated by the robot. In this variant, we use a general antipodal grasp sampler [Nguyen, 1985] and RRT-Connect [LaValle and Kuffner Jr, 2001] for generating robot trajectories, of which we utilize MPlib[‡] as the implementation. For *Hooking*, we also have a variant where there is only a single tool object provided to the robot (no tool selection). For all tasks, we only provide a single demonstration of object motion trajectories, the canonical view, and the pair of contact points on reference objects.

Baselines. We implemented two sets of baselines: global shape-matching and local shape-matching methods. For global shape-matching methods, we use two different methods: principal component analysis (PCA) and iterative closest point with FPFH features (FPFH+ICP) [Rusu *et al.*, 2009; Arun *et al.*, 1987; Zhou *et al.*, 2018] on object point clouds to find the best transformation that would align the target object and the reference object. Then, we retarget the object motion or transform the waypoints. For local shape-matching baselines, we also implemented two methods: *DINO matching* which directly finds the best functional correspondence based on the DINOv2 features, and *curvature matching* which uses the random contact point sampling method from Section 10.2.1 and applies the curvature filtering algorithm from MAGIC to rank all contact points. Since this method involves random sampling of contact points and simulation, we cap its runtime to 180 seconds and use the contact point with the highest matching score found at that time. Note that all methods except for the sampling-based curvature-matching algorithm have very small variances across different runs (our algorithm is completely deterministic). Therefore, we do not include performance variances for different methods. We do not compare with methods such as NDF [Simeonov *et al.*, 2022a], KETO [Qin *et al.*, 2020], and GIFT [Turpin *et al.*, 2021] in the main experiment, because they assume access to a collection of 3D object

[‡]https://github.com/haosulab/MPlib

Method	Scooping		Hanging		Hooking		
	FG	Arm	FG	Arm	No Tool Sel.	FG	Arm
PCA	27.6	16.7	0.0	0.0	5.0	0.0	0.0
FPFH+ICP	32.9	27.2	0.0	0.0	6.7	0.0	0.0
DINO Matching	55.7	41.4	23.9	21.6	49.2	66.7	24.0
Curvature-Only	13.9	5.6	68.7	38.1	60.0	76.7	3.3
MAGIC	**94.4**	**65.4**	**92.9**	**84.3**	**65.0**	**100**	**63.3**

Table 10.4: **Average success rates (%) on the scooping, hanging, and hooking tasks.** *floating gripper* (FG) indicates manipulation by directly setting positions and velocities of objects, focusing only on object-object interactions, excluding the robot. All methods except for Curvature-Only run almost deterministically. Curvature-Only has an average std. of 6.6%.

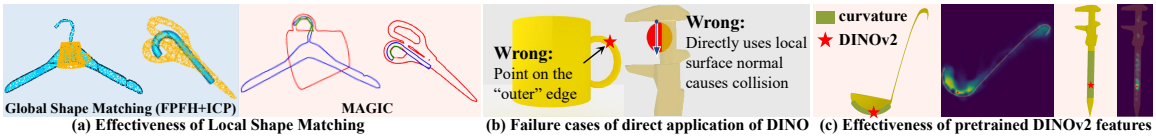


Figure 10-11: **Qualitative examples of contact point correspondence matching. (a)** Global shape matching fails to provide effective functional alignment due to the lack of awareness of local contact points. **(b)** Local curvature helps identify more accurate contact points and contact normals. **(c)** Pretrained DINOv2 features enhance data efficiency by selecting the optimal contact (the red star) and grasping points from those that curvature alone cannot distinguish (the green region). We also visualize the DINOv2 matching heatmaps.

models for representation learning, but we provide a case study of comparing MAGIC with KETO in the following section.

10.5.6 Simulation Experiments

Table 10.4 summarizes the overall performances of variants of our methods and other baselines. The success rates are computed over 6 spoons and 3 objects for *Scooping*, 134 mugs for *Hanging* adopted from [Simeonov *et al.*, 2022a], and 4 tools and 5 objects for *Hooking* adopted from Section 10.4. Overall, our model MAGIC performs the best. We break down our analysis into the following bullet points.

Local shape matching can significantly improve performance over global shape matching. We compare MAGIC with global shape matching methods, PCA and FPFH+ICP. As shown in Table 10.4, global shape matching is ineffective because it lacks an understanding of the local contact points. Fig. 10-11(a) illustrates an example of the correspondence

	PCA	FPFH+ICP	DINO Match	Curvature-Only	MAGIC
Hanging (intra-cat.)	5.2	41.1	64.9	67.9	**94.0**
Hanging (inter-cat.)	0.0	0.0	23.9	68.7	**92.9**

Table 10.5: **Average success rates (%) of the hanging task (intra-category transfer vs. inter-category transfer).** While PCA and FPFH+ICP can occasionally achieve intra-category transfer, they fail to perform inter-category transfer. The experiments are conducted on the *FG* variant.

established by global shape matching (left) and MAGIC (right). To further demonstrate the effectiveness of local shape matching, we also provide an additional study on the mug-hanging task in Table 10.5, where the demonstration includes the trajectory of hanging a held-out mug on the mug tree (i.e., intra-category transfer), rather than the hanger. In this setting, FPFH+ICP achieves a success rate of 41.1% (*c.f.* 94.0% for MAGIC). The results indicate that PCA and FPFH+ICP (as well as the DINO matching algorithm) can sometimes successfully achieve intra-category transfer by aligning overall shapes. However, they perform poorly at inter-category transfer due to the lack of local contact point alignment. In contrast, MAGIC excels in both intra-category and inter-category transfers by accurately identifying contact points that satisfy both global and local constraints.

Local curvature matching is crucial, in addition to pretrained features. The comparison between MAGIC and other methods that do not use local curvature information (e.g., PCA, FPFH+ICP, and DINO Match) shows the importance of local curvature matching. As an illustration, as shown in Fig. 10-11(b), the contact points proposed solely by the pretrained model are likely to fail due to incorrect curvature sign and incorrect interaction direction. In contrast, MAGIC leverages local curvature information to identify the appropriate contact points and correct force direction through the curvature estimation. Moreover, on the *Hooking* task, pairwise curvature matching plays an important role in selecting the most appropriate tool for different objects, as shown in the comparison between the *no-tool-selection* and the *FG* variant in the hooking task.

Using pretrained features as guidance significantly improves contact point selection. In Table 10.4, MAGIC shows superior performance over the sampling-based curvature matching algorithm. In principle, without noise in curvature computation and given a sufficient number of sampled points, the curvature-based method would be capable of finding the best contact. However, it will be very slow (e.g., in Section 10.4, we set the timeout to 600 seconds for the search). In this section, we leverage the global and semantic correspondence provided by DINOv2 features to reduce the searching space of local geometry matching. In

Method	Scooping	Hanging	Hooking	Avg. Time
MAGIC (DIFT + Curvature)	62.2	81.3	**100**	60.8s
MAGIC (SD-DINO + Curvature)	73.8	91.8	96.0	50.7s
MAGIC (DINOv2 + Curvature)	**94.4**	**92.9**	**100**	**34.0s**

Table 10.6: **Ablation of the choice of pretrained image features.** We substitute DINOv2 with SD-DINO [Zhang *et al.*, 2023] and DIFT [Dosovitskiy *et al.*, 2021]. We present the average success rates (%) and average time consumed of the methods. The experiments are conducted on the *FG* variant.

Fig. 10-11(c), we visualize how pretrained DINOv2 features can improve data efficiency by identifying the best contact and grasping points that curvature alone cannot differentiate.

Moreover, we provide additional comparison of different pre-trained features such as DIFT [Tang *et al.*, 2023] and SD-DINO [Zhang *et al.*, 2023] in Table 10.6. The results indicate that DINOv2 not only provides better guidance for shape matching but it also runs faster. This difference can be attributed to the distinct pretraining tasks and model architectures of DINOv2. DINOv2 employs self-supervised learning techniques, involving contrastive learning and a teacher-student training paradigm, where the teacher network has access to global views and distills knowledge to the student network — which has access to both global and local views [Oquab *et al.*, 2024]. This approach enhances the model's understanding of global information. In contrast, diffusion models are trained on generation tasks through a denoising process, which focuses more on local textures [Nichol and Dhariwal, 2021]. Regarding model structures, DINOv2 uses Vision Transformer (ViT) [Dosovitskiy *et al.*, 2021] with an attention mechanism applied to patches of the input image, whereas diffusion models are based on U-Net [Ronneberger *et al.*, 2015] and process the input and output as full-resolution images, requiring more computation time.

Further comparison with dataset-dependent methods. KETO [Qin *et al.*, 2020] and GIFT [Turpin *et al.*, 2021] introduce keypoints as the guidance for tool-using strategies and utilize self-supervised learning on the positive samples obtained from trial-and-error with the tools from a large tool dataset. While MAGIC also introduces a similar concept, contact points, we propose to consider contacts as pairs of contact points. In contrast, keypoints only focus on the tool object. Moreover, both KETO and GIFT need a dataset of tools, while MAGIC only requires one single demonstration and can be generalized to tools with very different shapes.

In Table 10.7, we provide additional experiment results comparing MAGIC with KETO over the *Hammering* and *Pushing* tasks from KETO. Since KETO requires a moderately large dataset of tool meshes to perform self-supervised learning, which is not available

Method	Hammering			Pushing
	Hammers → Hammers	Hammers → Non-hammers	Average	Average
KETO	66.4	46.3	56.4	71.7
MAGIC	**75.6**	**53.7**	**64.4**	**76.3**

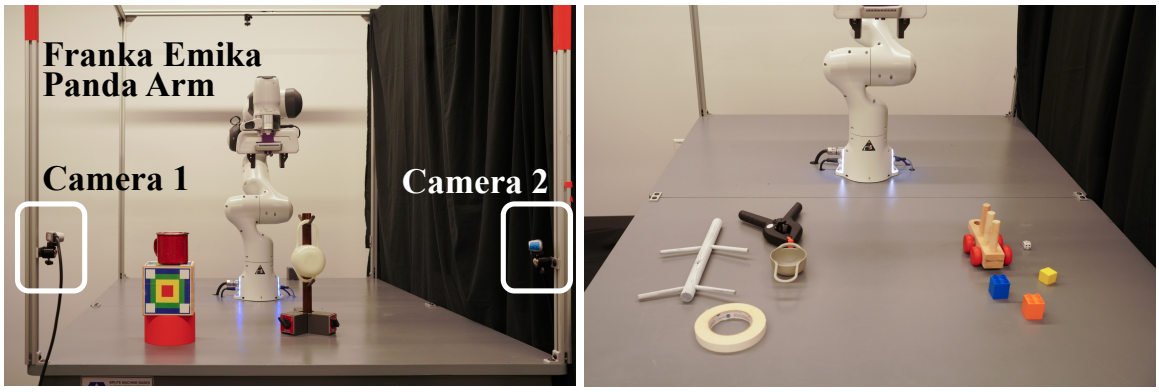
Table 10.7: **Comparison between MAGIC and KETO [Qin *et al.*, 2020] (Learning Keypoint Representations for Tool Manipulation).** We present the average success rates (%). We also investigate the intra-category and inter-category generalizablity of MAGIC and KETO, indicated by Hammers → Hammers and Hammers → Non-hammers, respectively.

Method	Scooping	Hanging	Hooking	Avg. Time
DINO Matching (Best Baseline)	55.7	23.9	66.7	**12.4s**
MAGIC (DINOv2 + Curvature)	**94.4**	92.9	**100**	34.0s
(w/o. Feature PCA and Patch)	71.7	**93.3**	62.0	33.8s
(w/o. Two-Step Curvature)	83.3	53.7	75.6	14.1s
(w/o. Simulation Verification)	77.8	71.6	50.0	19.7s

Table 10.8: **Ablation studies of different modules in MAGIC.** We present the average success rates (%) and average runtime of all methods, on the *FG* variant.

for those tasks in our main experiment, we perform experiments on their tasks. The single demonstration for MAGIC is composed of an image of a reference hammer taken in the simulator and the pixel coordinates of the function point and the effect point. The results indicate that MAGIC outperforms KETO on both tasks. To further investigate the generalizability, we provide the success rate of transferring from hammers to hammers and non-hammers, respectively. For both intra-category and inter-category generalization, MAGIC performs better than KETO.

Ablation studies. Table 10.8 provides additional ablation studies on different design choices of our method MAGIC. First, there is a significant performance drop observed in the scooping and hooking tasks if we remove feature PCA and patch-based score aggregation. This demonstrates that both methods reduce noise in the feature maps and improve the accuracy of contact point matching. Second, we consider removing the irrelevant point suppression and convexity matching. The model still surpasses vanilla DINO matching, which demonstrations the effectiveness of curvature estimation, but we see a clear performance drop compared with the full algorithm MAGIC. Finally, we show that removing simulation verification significantly degrades the performance.



(a) Hanging (b) Hooking

Figure 10-12: **Real-world experiment setup.**

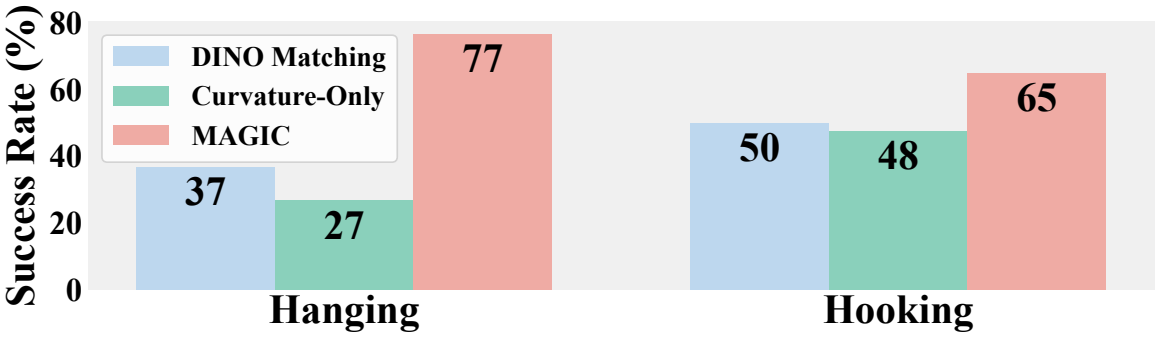


Figure 10-13: **Average success rates of real-world experiments.**

10.5.7 Real-World Experiments

To evaluate the effectiveness of MAGIC in real-robot systems, we conducted experiments on *Hanging* and *Hooking* utilizing a Franka Emika Panda Arm with a parallel gripper, as depicted in Fig. 10-12. We capture the RGB and depth image of the scene with two RealSense D435 cameras, whose extrinsics are calibrated to the frame of the base link of the robot.

We perform contact point matching on the RGB image captured by the camera, then import a reconstructed (possibly partial) object mesh from the RGBD camera into the simulator SAPIEN 2 for motion retargeting, motion planning, and verification. To reconstruct the mesh of the objects from the RGBD image, we first use the Segment Anything model [Kirillov *et al.*, 2023] to get the mask of the objects, and extract the corresponding point cloud of each object. Then, we remove outliers with DBSCAN [Ester *et al.*, 1996] and select the cluster with the largest number of points. We then perform object completion by projecting the

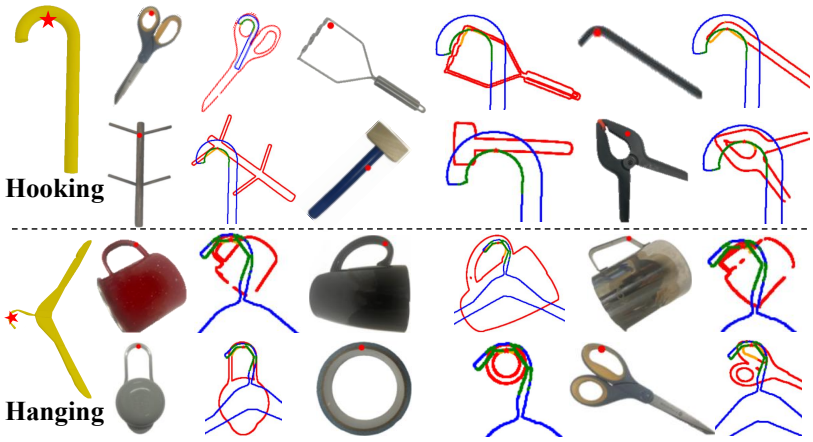


Figure 10-14: **Contact point matching results on real-world objects.**

point cloud down to the table plane, and using Alpha Shape [Edelsbrunner *et al.*, 1983] to reconstruct the surfaces of the object mesh.

To provide a quantitative assessment, we executed a total of 10 trials for each object. We report the success rates of various methods for *Hanging* across three visually distinct cups and *Hooking* across four tools from different categories, as shown in Fig. 10-13. MAGIC has consistently achieved the highest performance, which validates the effectiveness and practical applicability of MAGIC in the real world. We also illustrate the contact point matching found by MAGIC on more real-world objects in Fig. 10-14.

10.6 Keypoint-Based Manipulation Strategy Learning

To enable learning of operations that are not prehensile pick and place or quasi-static pushing, it is necessary to observe and remember some aspects of a motion trajectory; the challenge in doing this is enabling generalization over viewpoints, object positions, and object shapes.

In this section, we introduce KALM (Keypoint Abstraction using Large Models), a framework that prompts large vision-language models to propose candidate task-relevant keypoints, verifies their consistency across a handful of demonstrations, and distills them into compact abstractions. These keypoints are then used as object-relative action frames for training a keypoint-conditioned diffusion policy. KALM enables robots to acquire versatile manipulation skills with minimal demonstrations, while generalizing effectively across variations in object shape, pose, and category.

To address these generalization challenges, one common strategy for improving data efficiency is to leverage abstractions. Researchers have explored object-centric [Zhu *et al.*, 2023b], part-centric [Liu *et al.*, 2023c], and keypoint-centric representations [Gao and Tedrake, 2021] for tasks spanning rigid-body manipulation to deformable object handling,

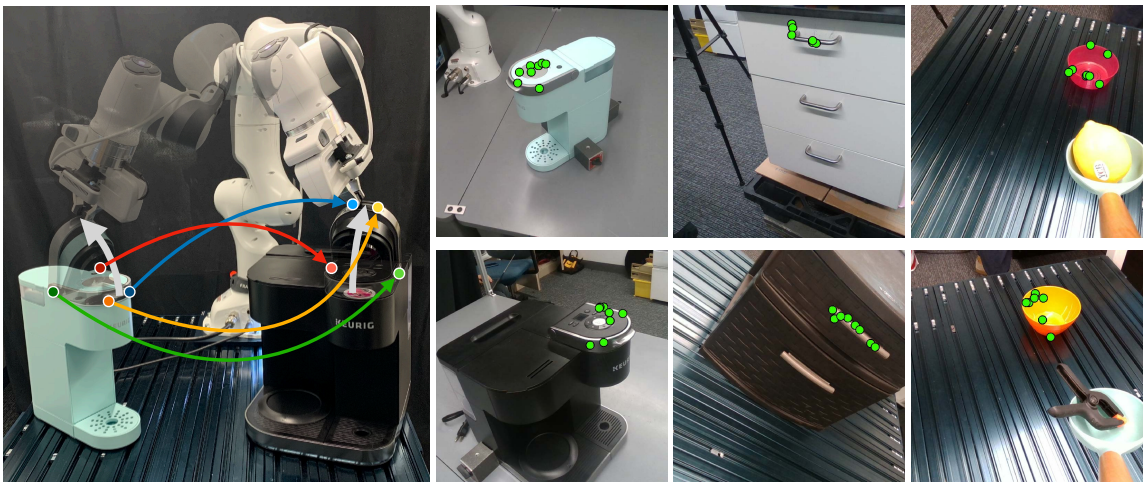


Figure 10-15: **Keypoint abstraction using large models for object-relative imitation learning.** KALM is a framework that distills keypoint abstraction by prompting and verifying keypoint proposals from large pre-trained models using a small amount of robot demonstration data, which is used to train a keypoint-conditioned policy model. Our method demonstrates strong generalization on multiple real-world manipulation tasks with only 10 demonstrations and no additional labeling effort.

such as ropes and clothes. Among these, keypoints offer a versatile abstraction for many robotic tasks, capturing essential object features with a low-dimensional encoding.

However, despite their potential for data efficiency and generalization, constructing a good keypoint representation can be tedious, both during training and at test time. Such training typically requires human experts to design task-specific keypoints, while deploying keypoint-centric models in the real world necessitates visual modules capable of detecting these keypoints, which often requires additional data collection and human annotations.

On the other hand, recent advances in large pre-trained models offer a promising alternative for the automatic construction of keypoint-centric representations, but their reliability remains a concern. Previous attempts to use these models for unsupervised keypoint extraction [Huang *et al.*, 2024a] have exclusively relied on the proposals of task-relevant objects or keypoints from pre-trained models and are therefore error-prone. Furthermore, it is unclear how these models, usually trained on little or no robotics data, can be aligned with the constraints and uncertainty in the physical environment. Our key insight is that an appropriate keypoint abstraction should satisfy two essential criteria: task relevance and cross-instance consistency. Task relevance ensures that the keypoints directly support the specific robotic task at hand. Cross-instance consistency ensures that these keypoints are robustly identifiable across multiple instances: from different camera views, over objects having similar functional shapes, in different poses. We hope to use a small amount of robot

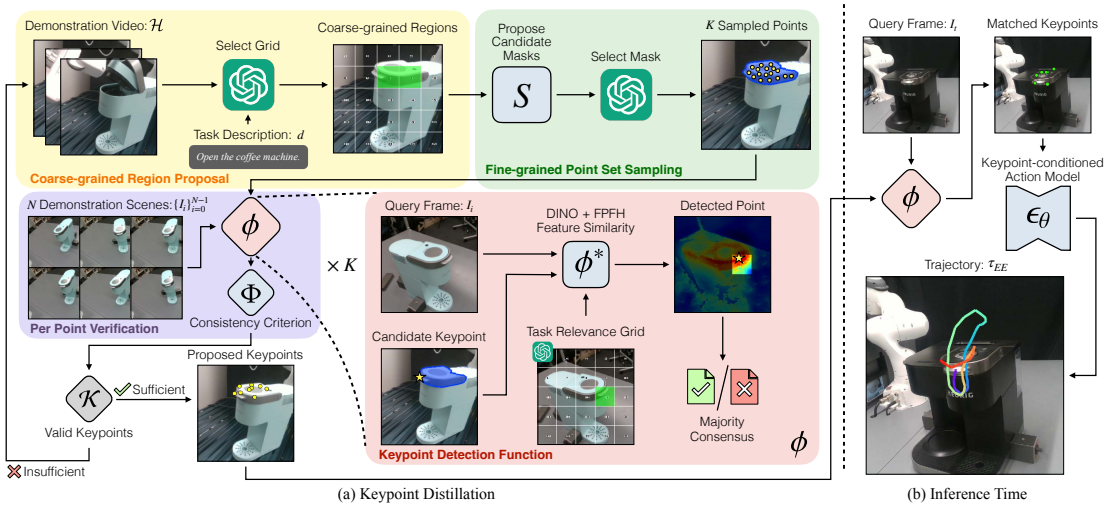


Figure 10-16: Overview of KALM.
(a) Keypoint distillation. Given a demonstration video and a task description, we prompt a VLM to generate a coarse-grained region proposal, which is refined into a fine-grained point set via image segmentation models and VLMs. We use a keypoint detection function ϕ to identify keypoint correspondences across a handful of demonstration trajectories. The final keypoints set is selected based on correspondence consistency verification. These keypoints are used for training a keypoint-conditioned action model. **(b) Inference time.** Given a new scene, the keypoint detection function ϕ localizes the distilled keypoints. The learned keypoint-conditioned action prediction model generates an object-relative end-effector trajectory based on the keypoint positions and features.

demonstration data (5 to 10 demonstrations) to align the large pre-trained models to the robotics domain.

In this section, we propose *Keypoint Abstraction using Large Models for Object-Relative Imitation Learning* (KALM), a framework that distills keypoints by prompting large pre-trained models and verifying the proposed keypoints based on a small number of robot demonstration trajectories. For each new task, given a single seeding demonstration video and a short natural language task description, we prompt a vision-language model to identify candidate object parts that are task-relevant. These candidates are further refined through querying image segmentation models into a candidate set of keypoints. The final keypoint set is selected based on cross-instance consistency computed across available demonstration trajectories of the task. Once the keypoints are identified, we train a diffusion policy model conditioned on these keypoints and their features to generate robot trajectories relative to the object keypoints. Our system demonstrates strong generalization across real-world settings, adapting to changes in environment and object instances.

Overall, our contributions are: First, we propose to distill task-relevant and cross-instance consistent keypoints from large pre-trained models with a combined proposal and

verification process. Second, with the extracted keypoints, we build a keypoint-centric, object-relative action representation based on keypoint features and their derived local frames that can be learned by diffusion policy models from a few demonstrations. Our keypoint-conditioned policy model allows robots to generalize their learned behaviors across different environments and object configurations, improving task performance under diverse conditions. Illustrated in Fig. 10-16, our proposed method distills keypoints through an iterative procedure, alternating between generating candidate proposals using large pre-trained models and verifying them based on a small robot demonstration dataset. Once the keypoints are distilled, we use their visual and geometric features to train a keypoint-conditioned diffusion model for generating object-relative robot actions. At inference time, our method detects the points corresponding to the previously distilled keypoints in the observed image, predicts the robot actions relative to the detected points, and finally transforms them back into the world frame for execution.

10.6.1 Problem Formulation

Formally, for each skill α, we require a single video of the robot successfully executing the task $\mathcal{H}^\alpha$, a handful of demonstration trajectories (5 to 10) $\mathcal{D}^\alpha$, and a natural language task description d^α. Each trajectory D_i^α contains an initial observation of the scene, represented as a calibrated RGBD image I_i^α and a robot joint trajectory τ_i^α. No additional labels such as keypoints are required.

Our first goal is to distill a set of keypoints that are task-relevant and consistent across observed images in all demonstration trajectories $\{I_i^\alpha\}$. We represent this keypoint set using 3D locations and their features in the first frame of the demonstration video, denoted as $\mathcal{K}^\alpha$. The second goal is to learn a trajectory prediction model, based on the distilled keypoints and demonstration trajectories $\mathcal{D}^\alpha$. In this section, we assume the trajectory of each skill is segmented into two distinct phases: an approaching phase, during which the robot moves freely towards the object, and an execution phase. The trajectory prediction model only predicts the execution trajectory, focusing only on the actions needed to manipulate the objects, since the approach trajectory can be generated via motion planning. The segmentation of the trajectory can be done through thresholding the distance to the closest keypoint, or through methods checking more fine-grained trajectory statistics [Ke *et al.*, 2024; Gervet *et al.*, 2023; Shridhar *et al.*, 2022b]. For brevity, in the following, we omit the superscript α when there are no ambiguities.

10.6.2 Keypoint Desiderata

In our framework, the keypoints serve both as an abstraction of the observational input and the basis of an object-relative action frame. To ensure that the abstraction is effective and the frame is robust to changes in the environment, we define two criteria: task relevance and cross-instance consistency.

Task relevance. To allow generalization to different scene configurations of the same task, the keypoints must be task-relevant. For example, for the task of lifting the handle of a coffee machine, the points on the handle are ideal candidates, whereas those on the water reservoir are not, because the latter vary across different machines and do not directly support the task completion.

Cross-instance consistency. Furthermore, it is essential that the keypoints are consistently identifiable across observations regardless of the object pose, camera view, or detailed shape of objects. For example, within a task-relevant object part, a point on the corner may be favored over one on a plain surface, due to its saliency and lower degree of ambiguity.

We evaluate the task relevance of a keypoint by leveraging pre-trained vision-language models, and check the cross-instance consistency using a keypoint detection function ϕ. Our goal is to find a set of keypoints that are both task-relevant and consistently identifiable.

10.6.3 Keypoint Proposal and Verification

Our keypoint proposal and verification pipeline works in three steps. First, we prompt a pre-trained vision language model (VLM) to select task-relevant image regions. Within the regions, we generate queries to image segmentation models to generate candidate object part masks, which are further ranked and selected by a second query to the VLM. We sample fine-grained keypoint proposals within the selected mask and score them based on consistency across all query images from the training demonstration set. This process will either return a set of final keypoints or declare failure, leading to another iteration of keypoint proposal. The overall process is illustrated in Figure 10-16a and Algorithm 8.

Coarse-grained region proposal $\boldsymbol{VLM}_1(\mathcal{H}, d)$. Our input to the VLM consists of a sequence of images $\mathcal{H}$ showing a single demonstration video of the robot executing the task, along with a natural language description of the skill d (e.g., "open the top drawer"). We aim to identify the regions of interest in the initial image, $I_0^{\mathcal{H}}$, associated with this video. We present $I_0^{\mathcal{H}}$ with an overlaid grid, where each grid cell is indexed by a unique text label, and query the VLM to select the grid indices corresponding to the task-relevant regions.

Algorithm 8 Keypoint Proposal and Verification Pipeline

Input: Skill description d, training set $\mathcal{D} = \{\langle I_i, \tau_i \rangle\}_{i=0}^{N-1}$, demonstration video $\mathcal{H} = \{I_t^{\mathcal{H}}\}_{t=0}^{|\mathcal{H}|-1}$
Output: Set of proposed keypoints $\mathcal{K}$ and their matched point in the training set $\mathcal{P} := \phi(k, I_i), \forall k \in \mathcal{K}, i \in I_i$

while True do
  $\mathcal{K} \leftarrow \emptyset$; $\mathcal{P} \leftarrow \emptyset$
  $\mathcal{K}_{VLM} \leftarrow VLM_1(H, d)$ ▷ Region proposal
  $\mathcal{M}_s \leftarrow \cup_{k \in \mathcal{K}_{VLM}} S(I_0^{\mathcal{H}}, k)$ ▷ Image segmentation
  $m \leftarrow VLM_2(I_0^{\mathcal{H}}, k, \mathcal{M}_S)$ ▷ Mask selection
  $\mathcal{K}_{FPS} \leftarrow FPS(I_0^{\mathcal{H}}, m)$ ▷ Fine-grained point sampling
  for each $k' \in \mathcal{K}_{FPS}$ **do**
    if $\frac{1}{N} \sum_{i=0}^{N-1} \mathbb{1}\left[\phi(k', I_i) \neq \textbf{Null}\right] \geq 1 - \delta$ **then**
      $\mathcal{K} = \mathcal{K} \cup \{k'\}$ ▷ Add k' to the output set
      $\mathcal{P} = \mathcal{P} \cup \{\phi(k', I_i) \mid \forall I_i\}$
  if $\frac{|\mathcal{K}|}{|\mathcal{K}_{FPS}|} \geq \gamma$ **then**
    break

In addition to the grid index, we employ zero-shot chain-of-thought [Wei *et al.*, 2022] prompting, encouraging the VLM to generate a textual description of the target object and its highlighted part before generating the final prediction.

Mask proposal $S(I_0^{\mathcal{H}}, k)$ and $VLM_2(I_0^{\mathcal{H}}, k, \mathcal{M}_S)$. Next, for each candidate coarse-grained region (represented as a grid cell in $I_0^{\mathcal{H}}$), we generate a uniformly distributed set of query points within the cell, and use a point-prompted image segmentation model [Kirillov *et al.*, 2023] to generate a set of object-part masks $\mathcal{M}_S$ for each of these query points. We apply standard non-maximum suppression to filter out masks with significant overlaps and discard those with low confidence scores.

Subsequently, we provide the conversation history from the previous VLM query, along with the image $I_0^{\mathcal{H}}$ overlaid with all the detected masks $\mathcal{M}_S$, as input of a second VLM query VLM_2. The VLM is tasked with selecting a single mask from $\mathcal{M}_S$ that contains the potential task-relevant keypoints. The output of this step is a mask m on the image $I_0^{\mathcal{H}}$. By leveraging the VLM's understanding of both the task context and the object-part segmentation masks, we obtain a more detailed representation of the task-relevant region.

Fine-grained Point Sampling $FPS(I_0^{\mathcal{H}}, m)$. Given the input RGBD image $I_0^{\mathcal{H}}$ and the selected mask m from the second VLM query, we apply Farthest Point Sampling [Eldar *et al.*, 1997] to the 3D points located inside the mask m to generate N_c candidate points $\mathcal{K}_{FPS}$. We found Farthest Point Sampling to be effective at generating diverse spatially distributed candidate keypoints that are geometrically salient, such as those located on part boundaries.

These points are often visually distinct and thus tend to be consistently identifiable. We will then select a set of keypoints from this set through our cross-instance consistency verification.

Keypoint detection function $\phi(k, I)$. Each candidate keypoint k in $\mathcal{K}_{FPS}$ is internally represented by its position p_0^k and feature vector $F_{ref}(p_0^k)$ in the initial scene $I_0^{\mathcal{H}}$. We need to detect their corresponding points in the training set observations $\{I_i\}$. We treat the keypoint detection task as a correspondence matching problem, where the goal is to identify the matching keypoint in a new input image I. The keypoint detection function can be written as $\phi(k, I) \to p$, where $I \in \mathbb{R}^{H \times W \times 4}$ is a query RGBD image, and $p \in \mathbb{R}^3$ is the 3D position of the keypoint in the query scene. When no matched keypoint is found in I, ϕ returns **Null**.

The matching accomplished using a scoring function ϕ^* based on the per-point feature representation in I. Formally, given the keypoint position p_0^k in $I_0^{\mathcal{H}}$, the function ϕ returns the position p in I that maximizes the following score:

$$\phi^*(p, p_0^k) = \langle F_q(p), F_{ref}(p_0^k) \rangle, \tag{10.1}$$

where $\langle \cdot, \cdot \rangle$ denotes the cosine similarity between two vectors. We also use the same VLM pipeline to obtain a coarse image grid of the target keypoint on the new input image I and discount points outside the mask in this matching process.

To enhance robustness, we implement a local group-based matching strategy that randomly samples $N' = 8$ neighboring points within a radius of $r = 0.02$ m around p_0, computes the corresponding match for each of them in the input image I, and declares a non-match if there is no majority consensus among the sampled points (i.e., if the majority of the matched points in I are not within close proximity of each other).

Cross-instance consistency verification. The previous steps generate a set of proposed keypoints $\mathcal{K}_{FPS}$ from the VLM and their corresponding points (or non-match) in each image of the demonstration trajectories. We evaluate each keypoint based on Φ that requires a successful correspondence matching in a majority of demonstration trajectories:

$$\Phi(k) := \frac{1}{N} \sum_{i=0}^{N-1} \mathbb{1} \left[\phi(k, I_i) \neq \textbf{Null} \right] \geq 1 - \delta, \tag{10.2}$$

where the acceptance factor $\delta = 0.3$ in our experiment.

Our success criteria for the proposal set $\mathcal{K}_{FPS}$ is that the majority of points in this set should be consistent candidates: $\sum_k \Phi(k)/|\mathcal{K}_{FPS}| \geq \gamma$. If there are sufficient consistent candidates in the proposal set $\mathcal{K}_{FPS}$, our algorithm returns those consistent candidates as our final selected keypoints $\mathcal{K}$. Otherwise, we will discard all proposal keypoints and

re-prompt VLM to generate another mask and set of proposal points. This step is important for verifying that the VLM selected part is both task-relevant and consistently identifiable.

10.6.4 Learning Keypoint-Centric Trajectory Prediction Models

From the keypoint proposal and verification process, we have determined a set of keypoints $\mathcal{K}$, which captures the most salient and task-relevant object parts for the skill α. For each demonstration trajectory D_i, we also have a set of corresponding points $\mathcal{P}_i$ from the previous keypoint detection step. Conditioned on the sparse keypoint locations and features of these detected points, we directly generate a trajectory τ_{EE} for the 6 DoF pose of the robot's end-effector using Diffuser [Janner *et al.*, 2022], a trajectory-level diffusion model.

We have two key design choices here to facilitate generalization: using a sparse keypoint-based input, and having actions predicted in a keypoint-centric, object-relative coordinate frame. Specifically, the model only takes the keypoint locations and their features as input, which leverages the keypoint abstraction to obtain invariance to task-irrelevant distractions, such as background and view changes. Meanwhile, we predict the actions relative to the center of these keypoints (and the orientation of the coordinate frame is fixed to be aligned with the robot base). The object-relative nature of the design makes our model invariant to changes in the absolute pose of the camera and the objects.

10.6.5 Implementation

We use GPT-4o [OpenAI, 2024] as our VLM and Segment-Anything Model [Kirillov *et al.*, 2023] (SAM) as our image segmentation model. In the similarity function ϕ^*, we use a combination of pre-trained image features (DINO [Oquab *et al.*, 2024] with FeatUp [Fu *et al.*, 2024]) and analytic 3D features Fast Point Feature Histograms (FPFH [Rusu *et al.*, 2009]). The overall (cosine) similarity in Equation 10.1 is defined as:

$$\begin{aligned}\langle F_q(p), F_{ref}(p_0^k)\rangle = \lambda_1 \cdot \langle F_q^{DINO}(p), F_{ref}^{DINO}(p_0^k)\rangle \\ + \lambda_2 \cdot \langle F_q^{FPFH}(p), F_{ref}^{FPFH}(p_0^k)\rangle,\end{aligned} \tag{10.3}$$

where $\lambda_1 = 0.75$ and $\lambda_2 = 0.25$ in our experiments.

For the trajectory prediction model, the trajectory τ_{EE} is represented as a sequence of $H = 48$ end-effector poses. The input to the Diffuser is an unordered set of keypoint features, represented as their DINO and FPFH features. We optimize the model using the standard diffusion loss function. The diffusion model estimates the conditional distribution $p(\tau_{EE} \mid \mathbf{C})$, where $\mathbf{C} = \{\langle p_k, F_k^{DINO}, F_k^{FPFH}\rangle\}_{k\in\mathcal{K}}$ represents the set of matched keypoints in D_i.

#Demos	Tasks	Without Keypoints		With Keypoints		
		RGB	RGBD	RGB	RGBD	
		Diffuser [Janner *et al.*, 2022]	3D Diffuser Actor [Ke *et al.*, 2024]	Diffuser	3D Diffuser Actor	**KALM (Ours)**
10	DrawerOpen	30.00 ± 8.29	30.00 ± 6.53	62.00 ± 1.41	29.33 ± 4.64	**77.00 ± 2.94**
	DrawerClose	50.00 ± 1.41	53.67 ± 4.19	83.33 ± 2.62	50.67 ± 6.60	**92.33 ± 0.47**
	ButtonSide	32.67 ± 2.49	37.67 ± 2.05	49.67 ± 11.09	38.67 ± 1.25	**79.67 ± 1.25**
	ButtonTop	19.00 ± 3.56	21.00 ± 4.90	28.00 ± 8.04	21.00 ± 4.32	**97.33 ± 0.47**
	LeverPull	10.67 ± 3.30	8.67 ± 2.36	21.33 ± 1.70	10.33 ± 4.19	**61.67 ± 6.13**

Table 10.9: **Few-shot learning results in Meta-World.** We evaluate our method, KALM, on five manipulation tasks in the Meta-World [Yu *et al.*, 2019] simulator, using Diffuser [Janner *et al.*, 2022] and 3D Diffuser Actor [Ke *et al.*, 2024] as baselines, along with ablation studies on keypoint usage. Our method consistently outperforms these baselines, demonstrating that keypoints serve as an effective abstraction.

10.6.6 Inference-time Pipeline

At inference time, given a new scene image I, we begin by running the keypoint detector $\phi(k, I)$ for all keypoints $k \in \mathcal{K}$, extracting their corresponding position and feature vectors. We then use the learned Diffuser to generate a set of end-effector trajectories. Note that the learned trajectory starts relatively close to the target object. We need to ensure reachability and collision-free motion in the environment. Similar to previous work that uses diffusion models as trajectory samplers [Fang *et al.*, 2024b], we use a motion planner to check whether there is a feasible path to the initial pose of the task trajectory $\tau_{EE}^{t=0}$. If a valid path exists, we move the end-effector to $\tau_{EE}^{t=0}$ using the approaching path computed by the motion planner, and start τ_{EE} execution, otherwise generate new ones.

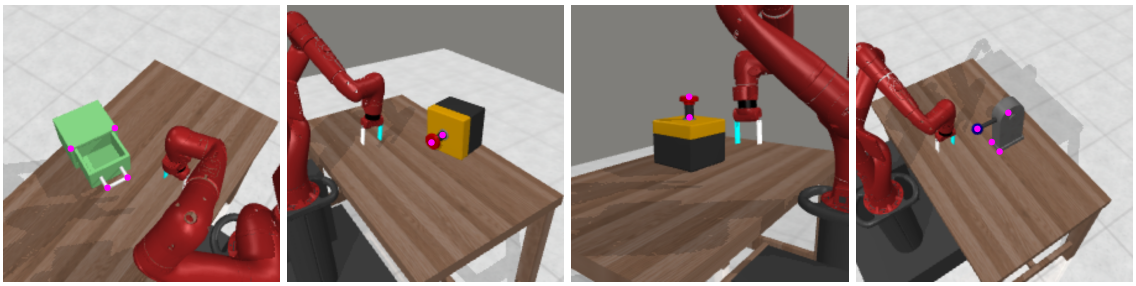
In this section, we want to study the following questions:

- Is the sparse keypoint abstraction sufficient for the conditional diffusion model to predict valid trajectories with only a limited amount of demonstration data?
- Does the keypoint abstraction improve data efficiency compared to the baselines?
- Does the iterative proposal and verification procedure distill appropriate keypoints for the action model learning?

We study the data efficiency of keypoint abstraction in a simulation environment due to the difficulty in collecting a large amount of data for training all baselines in the real world. We also compare different keypoint proposal methods in the real world by measuring the success rate and generalization of the keypoint-conditioned policies.

10.6.7 Simulation Experiments in Meta-World

In this section, we compare our method with other baselines with different input spaces and network architectures using the Meta-World [Yu *et al.*, 2019] simulator, focusing on the



(a) DrawerOpen DrawerClose (b) ButtonSide (c) ButtonTop (d) LeverPull

Figure 10-17: **Testing tasks in Meta-World [Yu *et al.*, 2019].** We evaluate on 5 tasks in Meta-World with randomized camera and object poses, necessitating the generalization of policies across observational changes. Keypoints are marked in pink for visualization.

efficacy of different representations in terms of their data efficiency.

Setup. We evaluate on 5 tasks: DrawerOpen, DrawerClose, ButtonSide, ButtonTop, and LeverPull, as shown in Fig. 10-17. To examine the role of input modality independently of keypoint detection accuracy, we employ an oracle set of keypoints in our simulation experiments. We manually label the XML files and compute the 2D or 3D keypoint locations in the scene given object and camera poses. We train and test on scenes with varying camera and object poses. The object poses are uniformly sampled on the table with 2D translations and rotations. The camera viewpoint is sampled around the object with randomized angle and distance with elevation in $[0, \frac{\pi}{2}]$, azimuth in $[-\frac{\pi}{2}, \frac{\pi}{2}]$, and distance in $[1.5, 2]$. We exclude camera viewpoints where the object in the initial observation is occluded by the arm.

Baselines. We compare our method against 4 baselines using different input modalities.

- **Diffuser [Janner *et al.*, 2022] (RGB)** generates a 6 DoF end-effector trajectory based on an initial RGB observation of the scene and the camera extrinsics, using a 2D convolutional network. We use DINO [Oquab *et al.*, 2024] visual encoder and finetune it at training time to handle the discrepancy between the simulator and real-world images on which DINO is trained.

- **3D Diffuser Actor [Ke *et al.*, 2024] (RGBD)** builds a 3D representation from pre-trained visual features (CLIP [Radford *et al.*, 2021]) and the point cloud. It generates actions with a diffusion model conditioned on the 3D representation.

- **Diffuser with keypoints**: We provide Diffuser with the 2D position of the keypoints as an additional input.

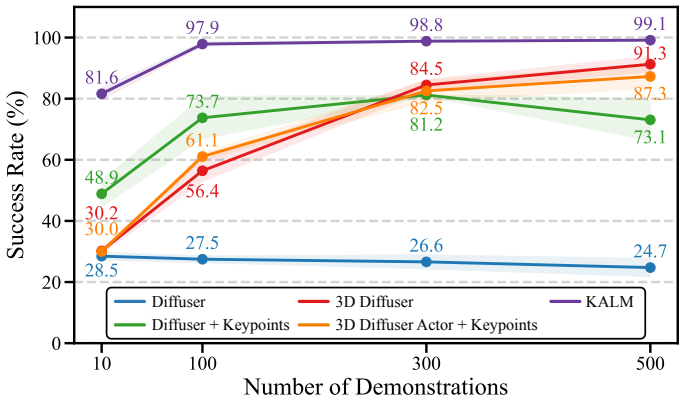


Figure 10-18: **Data efficiency results on Meta-World.** We measure the average success rate across all 5 tasks, with the number of demonstrations increasing from 10 to 500. Our method, KALM, demonstrates superior data efficiency compared to all baselines.

- **3D Diffuser Actor with keypoints**: Similarly, we provide the 3D position of the keypoints to the 3D Diffuser Actor.

Few-shot learning results. We present the task success rates of all baselines across 5 tasks trained with only 10 demonstrations in Table 10.9. The success rate is averaged over 100 test episodes and the variances are computed across three random seeds. Overall, the Diffuser (RGB) model struggles to predict accurate 3D trajectories. Nonetheless, we observe a significant performance improvement with the keypoints added. The 3D Diffuser Actor, which employs a transformer backbone, requires more training data. It has poor performance in low-data regimes. By contrast, our keypoint-conditioned diffusion model, taking only sparse inputs of 3D keypoint positions and their visual features, can learn efficiently from very few demonstrations while achieving strong performance.

Data efficiency study. We further evaluate the data efficiency by varying the number of demonstrations from 10 to 500. We report the average success rate on 5 tasks in Fig. 10-18 (each with three seeds and 100 test episodes). Our method KALM achieves superior data efficiency, reaching peak performance using only 100 demonstrations, whereas the 3D Diffuser Actor requires 500 demonstrations to achieve competitive performance. This validates the effectiveness of sparse keypoint abstraction for trajectory prediction.

10.6.8 Real-World Experiments on Franka Arm

In this section, we investigate whether our proposed iterative procedure generates better keypoints which eventually lead to a higher task success rate for a range of different tasks in the real world. We also explore the generalization capability along different dimensions



(a) Lifting the handle (b) Opening the drawer (c) Pouring into a bowl

Figure 10-19: **Testing tasks in the real world.** We evaluate different methods on three tasks in the real world with different objects at different poses, and with different camera angles. The testing assets are illustrated in the figure.

	w/o. Verification		KALM (Ours)	
Tasks	View	Cross Obj.	View	Cross Obj.
Lifting Handle	1/10	0/10	9/10	6/10
Opening Drawer	2/10	0/10	6/10	7/10
Pouring into Bowl	6/10	2/10	8/10	6/10

Table 10.10: **Results on real-world experiments.** We evaluate all models in varying camera and object poses (View), as well as on unseen objects (Cross obj.). Our method demonstrates strong performance under view changes and showcases generalization to objects that are not seen during diffusion training time.

endowed by using keypoint abstraction such as object poses and object instances.

Setup. We carry out real-world experiments on three tasks: 1) Lifting the handle of a coffee machine, 2) Opening the top drawer, and 3) Pouring something from a spoon into a bowl, as illustrated in Fig. 10-19. We use a Franka Research 3 robot arm with a RealSense D435f RGBD camera mounted on the gripper. For each task, we collect 10 demonstrations and capture the initial images using the gripper-mounted camera.

To evaluate the generalization along different dimensions, we vary the environment in terms of camera and object poses (View), and object instances (Cross obj.). Note that the objects are never seen during the training of the diffusion model, where changes in camera and object poses are also applied.

Baseline. Since collecting sufficient data to train the Diffuser and 3D Diffuser Actor for robust real-world performance across different views and camera poses is challenging, we instead focus on a few-shot setting. In this setup, we specifically validate our keypoint

proposal pipeline, which is not evaluated in the simulation environment. To validate the efficacy of the verification procedure, we compare our method against a baseline that directly prompts VLMs to select a set of keypoints without performing cross-instance consistency verification, referred to as "w/o. Verification". Specifically, we make another query to VLM after obtaining the fine-grained point sampled $\mathcal{K}_{FPS}$, and ask it to select $N_k = 8$ keypoints, as the final output.

Few-shot learning results. We carry out 10 repeated runs for each task. For a fair comparison, we run the baseline and KALM with the same set of 10 setups: the robot is reset to the same initial pose before executing the predicted trajectory, and we reset the objects as close as possible if they were moved by the robot. Table 10.10 shows the results.

Overall, our method consistently outperforms the baseline, highlighting the importance of cross-instance consistency verification. We observe that the baseline struggles with localizing corresponding keypoints in testing scenes, due to worse keypoint selections. Even when the keypoint detection is accurate, the predicted trajectories are significantly worse. We attribute the reason to the poor detection during keypoint distillation, which leads to a lower-quality dataset $\mathcal{P}$ for training the keypoint-conditioned action prediction model.

With the object-relative action representation, our method achieves strong performance under view changes, abstracting away the absolute position of the camera or object. This enables our method to perform strongly with only 10 demonstrations and no additional labels. We believe this is an efficient way to scale up generalizable skill learning.

10.7 Related Work

Task and motion planning for robot manipulation. Approaches toward manipulation with primitives have been extensively studied in the field of robotics. These approaches aim to design algorithms that can perform complex tasks by decomposing manipulation tasks into a sequence of primitive actions, such as grasping, pushing, and placing. Task and motion planning algorithms provide methods for generating high-level task plans while considering the robot's physical constraints and the environment's dynamics. These approaches can be roughly grouped into two groups: sampling-based such as HPN [Kaelbling and Lozano-Pérez, 2011], IDTMP [Dantam *et al.*, 2018], and many others [Dogar and Srinivasa, 2012; Srivastava *et al.*, 2014; Garrett *et al.*, 2020a], and global optimization-based such as Logic-Geometric Programming [Toussaint, 2015] and Pang *et al.* [2023]. Our method falls into the group of sampling-based planning, and we extend existing approaches to handle briefly-dynamic tasks [Mason, 2001]. The major difference between our work and existing work in task and motion planning is that we focus on learning *mechanisms* that are sequences of

basis manipulation operations and show significant improvement in complex manipulation tasks.

Contact-based modeling for robotics. Our algorithm draws inspiration from contact-based modeling approaches in robot manipulation. In particular, various methods have been presented to make manipulation plans in the contact space, between rigid bodies as well as within robot hands [Trinkle and Hunter, 1991; Ji and Xiao, 2001; Yashima *et al.*, 2003; Lee *et al.*, 2015; Mordatch *et al.*, 2012; Hauser and Latombe, 2010; Sleiman *et al.*, 2019; Aceituno-Cabezas and Rodriguez, 2020; Huang *et al.*, 2023a]. Algorithms such as CMGMP [Xiao and Ji, 2001; Cheng *et al.*, 2021b, 2022] can automatically generate possible contact modes based on three primitive modes: fixed contact, separating, and sliding contact. In contrast to these methods that mostly focus on the characterization of contacts and planning with them, we consider learning *mechanisms* from contact modes in a single demonstration to solve new tasks efficiently.

Tool-using skills. Some mechanisms acquired by our model have been traditionally described as *tool-use* skills [Stoytchev, 2005; Bril *et al.*, 2010; Nabeshima *et al.*, 2007; Gonçalves *et al.*, 2014]. In comparison to these works, we presented a novel framework for generating tool use trajectories with a planner based on contact sampling and showed that the learned mechanisms can be recombined with a general task and motion planner to solve more complex tasks. Recently, people have explored learning to predict tool-using trajectories or "object affordances" [Gupta and Davis, 2007; Zhu *et al.*, 2015b; Fang *et al.*, 2020; Li *et al.*, 2022; Lai *et al.*, 2021; Mo *et al.*, 2021], usually from a given set of demonstrations. By contrast, this chapter focuses on generating trajectories with novel objects based only on a single demonstration, and combining the learned skills.

Object affordances. Recently, researchers have explored learning object affordances [Gupta and Davis, 2007; Zhu *et al.*, 2015b; Fang *et al.*, 2020; Qin *et al.*, 2020; Lai *et al.*, 2021; Mo *et al.*, 2021; Ju *et al.*, 2024], usually from a large set of demonstrations. In contrast, this chapter focuses on generating trajectories with novel objects from a single demonstration leveraging pretrained visual features and shape analysis algorithms, requiring no additional training data.

Shape representations for actions. For many years, people have attempted to harness shape information to guide tool usage through mathematical analysis [Asada and Brady, 1986; Jensen *et al.*, 2002; Brandi *et al.*, 2014; Rodriguez and Behnke, 2018; Biza *et al.*, 2023] and data-driven approaches [Simeonov *et al.*, 2022a; Thompson *et al.*, 2021; Manuelli

et al., 2019; Gao and Tedrake, 2021; Turpin *et al.*, 2021; Wen *et al.*, 2022]. These methods often necessitate intricate human specifications or extensive in-domain datasets. In contrast, our approach integrates the off-the-shelf visual models, pretrained on general image datasets, with the generic geometrical property of curvature. This combination allows us to achieve effective and efficient tool-using guidance with minimal human intervention and without the need for large specialized datasets.

Abstractions for action representations. Abstractions over raw sensorimotor interfaces have been shown to enhance data-efficient learning and facilitate generalization. Various forms of abstractions have been studied, including contact points [Trinkle and Hunter, 1991; Ji and Xiao, 2001; Lee *et al.*, 2015; Cheng *et al.*, 2022], objects [Diuk *et al.*, 2008; Devin *et al.*, 2018; Veerapaneni *et al.*, 2020; Wang *et al.*, 2021; Yuan *et al.*, 2021; Mao *et al.*, 2022; Zhu *et al.*, 2023b], object parts [Aleotti and Caselli, 2011; Vahrenkamp *et al.*, 2016; Myers *et al.*, 2015; Liu *et al.*, 2023c], keypoints [Manuelli *et al.*, 2019; Gao and Tedrake, 2021; Qin *et al.*, 2020; Turpin *et al.*, 2021], and other shape representations [Wen *et al.*, 2022; Simeonov *et al.*, 2022a; Shen *et al.*, 2023]. These spatial abstractions can serve as direct inputs to data-driven models [Wang *et al.*, 2021; Gao and Tedrake, 2021; Zhu *et al.*, 2023b], or be used to create coordinate frames [Niekum *et al.*, 2012; Liu *et al.*, 2023c]. Typically, these models represent short-horizon robot behaviors and can be sequentially composed [Qin *et al.*, 2020; Wang *et al.*, 2024b] in a fixed order, or integrated into high-level planning algorithms [Garrett *et al.*, 2021]. Our work leverages keypoint-based representations for few-shot imitation learning, and focuses on acquiring such representations automatically without additional data and labels.

Finding keypoint correspondences. Finding functional correspondences between objects [Lai *et al.*, 2021] has been extensively explored in robotic manipulation, using both analytical [Asada and Brady, 1986; Brandi *et al.*, 2014; Rodriguez and Behnke, 2018; Biza *et al.*, 2023] and data-driven methods [Simeonov *et al.*, 2022a; Thompson *et al.*, 2021; Manuelli *et al.*, 2019; Gao and Tedrake, 2021; Turpin *et al.*, 2021; Wen *et al.*, 2022]. However, the keypoints derived from these approaches are typically tailored to specific tasks, requiring human annotations at training time, or additional object datasets. In contrast, our method eliminates the need for human-labeled keypoints by leveraging large pre-trained models for vision-language grounding and visual recognition, enabling automatic discovery of task-relevant and cross-instance consistent keypoints.

Vision-language models for robotics. A large body of research has focused on utilizing pre-trained language and vision-language models for robotic manipulation, by generating

action plans [Ahn *et al.*, 2022; Liang *et al.*, 2023], specifying motion constraints [Huang *et al.*, 2023b, 2024a], and directly predicting robot movements [Hu *et al.*, 2023; Fang *et al.*, 2024a]. The primary limitation of these methods is their dependence on primitive sets of low-level controllers or on motions that can be easily described by simple analytical expressions. Additionally, because these approaches rely on one-shot solutions generated by pre-trained models, they are prone to errors produced by these models. Other work proposes generating reward functions [Ma *et al.*, 2024] or planning models [Liu *et al.*, 2021a] and can learn new policies from data, but they do not tackle the issue of data efficiency.

10.8 Discussion

This chapter has explored how robots can acquire, generalize, and reuse manipulation strategies by grounding them in structured abstractions of contact and geometry. Across three complementary works, we have moved progressively from defining reusable skills, to enabling their one-shot generalization, and finally to discovering richer constraints that capture the compositional structure of objects and tasks.

The first contribution introduced the concept of mechanisms: reusable skills expressed as sequences of contact mode changes, paired with mechanism-specific samplers. Mechanisms serve as macros that compress long planning horizons into abstract, symbolic units, allowing robots to efficiently recombine them into complex solutions. This formulation highlights the power of explanation-based learning in robotics—capturing the causal structure of contact interactions from a single demonstration and amplifying it through self-play.

Building on this, the second contribution is MAGIC, a framework for generalizing manipulation strategies to novel tools and objects without retraining or costly search. By treating manipulation as an analogy over contact waypoints, MAGIC leverages pretrained visual features for global shape correspondence and local curvature analysis for physical plausibility. This hybrid analytic–data-driven approach demonstrates that robust generalization can emerge not from memorizing object categories but from reasoning about functional contact affordances.

Finally, the third contribution is KALM, a framework for learning more complex contact constraints via keypoint abstraction. By prompting and verifying large pre-trained models with a handful of robot demonstrations, KALM distills task-relevant and cross-instance consistent keypoints, which serve as object-relative reference frames for policy learning. This approach enables data-efficient imitation learning, allowing robots to extrapolate across object instances, poses, and even categories.

The central theme throughout is that abstraction, whether in the form of contact sequences, functional correspondences, or task-relevant keypoints, is essential for bridging

low-level perception and control with high-level reasoning and planning. Looking forward, this perspective suggests new opportunities: integrating mechanism learning with language-guided reasoning, extending analogy-based generalization to multi-contact and multi-agent scenarios, and scaling keypoint abstractions into richer ontologies of parts, forces, and constraints.

In the next chapter, we will build on top of this foundation to tackle challenges in long-horizon manipulation problems in robotics.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 11

Neuro-Symbolic Long-Horizon Planning

This Chapter presents a framework for learning for long-horizon planning. It leverages two key ideas from two early chapters: neuro-symbolic visual concept learning introduced in Chapter 6, and neuro-symbolic composable actions introduced in Chapter 10, to build a planning-compatible representation of actions and novel neuro-symbolic planning algorithms. In particular, in addition to representing the mode-sequence structures of skills, we propose to also acquire the pre- and post-conditions of individual skills and plan with them.

We first introduce PDSketch, a planning-domain specification language and a corresponding planning algorithm, designed to balance structured inductive biases (locality, sparsity, and object-centrism) with learned neural modules. By exposing high-level structural priors while leaving lower-level functions to be learned, PDSketch achieves data-efficient model learning and enables domain-independent planning heuristics. Experiments in BabyAI and robotic painting domains demonstrate strong generalization to unseen goals and states.

While PDSketch assumes domain structures are specified by humans, many real-world domains demand automatic discovery of these structures. This motivates the BLADE framework (Behavior from LAnguage and Demonstration). Rather than relying on hand-specified domains, BLADE leverages language-annotated demonstrations together with large language models to automatically invent domain abstractions. Compared to PDSketch, BLADE focuses on propositional preconditions and effects, and introduces an extended training algorithm for more efficient grounding of LLM-invented symbols. By segmenting demonstrations into contact-based primitives, querying LLMs for candidate precondition–effect structures, and grounding them in perception, BLADE constructs behavior representations that enable long-horizon planning and generalization.

The material of this chapter has been previously published in Mao *et al.* [2022] and Liu *et al.* [2024b]. Weiyu Liu and Neil Nie, in particular, contributed significantly to the materials presented in this chapter.

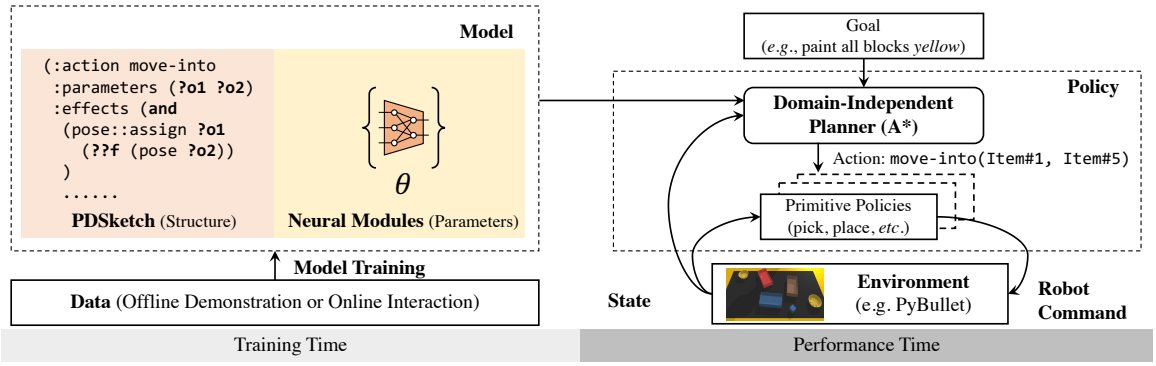


Figure 11-1: **The life cycle of a PDSketch model.** A PDSketch model is composed of a model structure definition and a collection of trainable neural modules. The model parameters can be learned from data. During performance time, the model is used by a domain-independent planner to form a policy that directly interacts with the environment.

11.1 Introduction

A long-standing goal in AI is to build robotic agents that are flexible and general, able to accomplish a diverse set of tasks in novel and complex environments. Such tasks generally require a robot to generate long-horizon plans for novel goals in novel situations, by reasoning about many objects and the ways in which their state-changes depend on one another. A promising solution strategy is to combine model learning with online planning: the agent forms an internal representation of the environment's states and dynamics by learning from external or actively-collected data, and then applies planning algorithms to generate actions, given a new situation and goal at performance time.

There are two primary desiderata for a system based on model-learning and planning. First, the learning process should be *data efficient*, especially because of the combinatorial complexity of possible configuration of in the real world. Second, the learned model should be *computationally efficient*, making online planning a feasible runtime execution strategy.

A critical strategy for learning models that generalize well from small amounts of data and that can be deployed efficiently at runtime is to introduce inductive biases. In image processing, we leverage translation invariance and equivariance by using convolutions. In graph learning, we leverage permutation invariance by using graph neural networks. Two essential forms of structure we can leverage in dynamic models of the physical world are *locality* and *sparsity*. Consider a robot picking an object up off a table. At the object level, the operation only changes the states of the object being picked up and the robot, and only depends on a few other nearby objects, such as the table (local). At the object feature level, only the configuration of the robot and pose of the object are changed, but their colors and frictional properties are unaffected (sparse).

```
def move_into(o, c):
  o.pose = c.pose + [0, 0, 0.1]
  if is_block(o) and is_painter(c):
    o.color = c.color
```

(a) A full specification of the transition model.

```
def move_into(o, c):
  o.prop1 = ??(c.prop1)
  if ??(o, c):
    o.prop2 = ??(c.prop2)
```

(b) A structure-only specification.

Figure 11-2: Defining both the transition model structure and implementation in Python (a) vs. defining only the structure while leaving details (the **??** functions) to be learned (b).

Classical hand-engineered approaches to robot task and motion planning have designed representations that expose and exploit locality through *lifting* (or *object-centrism*), which allows relational descriptions of objects and abstraction over them, and through *factoring*, which represents different attributes of an object in a disentangled way [Garrett *et al.*, 2021]. These representations are powerful and effective articulations of locality and sparsity, but they are traditionally laboriously hand-designed in a process that is very difficult to get correct, similar to writing a full state-transition function as in Fig. 11-2a. This approach is not directly applicable to problems involving perception or environmental dynamics that are unknown or difficult to specify. In this chapter, we present PDSketch, a model-specification language that integrates human specification of structural sparsity priors and machine learning of continuous and symbolic aspects of the model. Just as human users may define the structure of a convolutional neural network in TensorFlow [Abadi *et al.*, 2016] or PyTorch [Paszke *et al.*, 2019], PDSketch allows users to specify *high-level* structures of the transition model as in Fig. 11-2b (analogous to setting the kernel sizes), and uses machine learning to fill in the details (analogous to learning the convolution kernels).

Fig. 11-1 depicts the life-cycle of a PDSketch model. PDSketch uses an object-centric, factored, symbolic language to flexibly describe structural inductive biases in planning domains (i.e., the model structure). A PDSketch model is associated with a collection of neural modules whose parameters can be learned from robot trajectory data that are either collected offline by experts or actively collected by interacting with the environment. During performance time, a PDSketch is paired with a domain-independent planner, such as A^*, and as a whole forms a goal-conditioned policy. The planner receives the environmental state and the trained PDSketch model, makes plans in an abstract action space, and invokes primitive policies that actually generate robot joint commands.

Compared to unstructured models, such as a single multi-layer perceptron that models the complete state transition monolithically, the structures specified in PDSketch substantially improve model generalization and data-efficiency in training. In addition, they enable the computation of powerful *domain-independent planning heuristics*: these are estimates of the cost-to-go from each state to a state satisfying the goal specification, which can be obtained from the structured transition model *without any additional learning*. They can be leveraged by A^* to efficiently plan for *unseen* goals, specified in a first-order logic language.

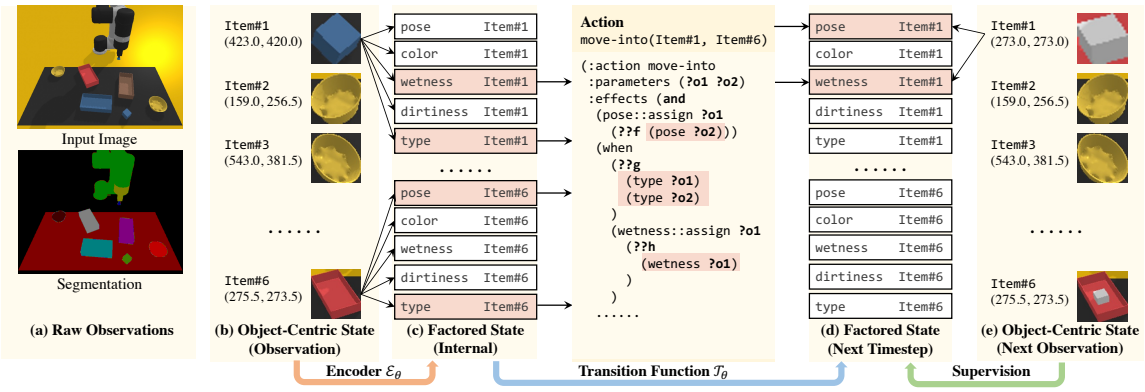


Figure 11-3: **Factorized state representations and transition models.** A factorized state representation and transition model for the robot painting domain. The raw observation (a) is first processed by an external perception module into an object-centric representation (b). This representation is further transformed into a fine-grained factorization (c). The transition function $\mathcal{T}_\theta$ can be defined over this factored state representation: each action may only change a few factors of the state. Executing a specific action `move-into(item#1, item#6)` produces the predicted factored state at the next timestep. During training time, we will be using the object-centric observation from the next timestep to supervise the learning of $\mathcal{T}_\theta$.

We experimentally verify the efficiency and effectiveness of PDSketch in two domains: BabyAI, a 2D grid-world environment that focuses on navigation, and Painting Factory, a simulated table-top robotic environment that paints and moves blocks. Our results suggest that 1) locality and sparsity structures, specified economically in a few lines of code, can significantly improve the data efficiency of model learning; 2) the model learning and planning paradigm enables strong generalization to unseen goal specifications. Finally, the domain-independent heuristics automatically induced from the structures dramatically improve performance-time efficiency, especially for novel goal specifications.

While PDSketch assumes domain structures are specified by humans, many real-world settings demand automatic discovery of these structures. This motivates BLADE (Behavior from LAnguage and Demonstration), which we will introduce in Section 11.4. Instead of relying on hand-specified domains, BLADE leverages language-annotated demonstrations and large language models to acquire domain abstractions automatically. By segmenting demonstrations into contact-based primitives, leveraging similar ideas from Chapter 10, querying LLMs for precondition–effect structures, and grounding them in perception, BLADE builds behavior representations that support long-horizon planning.

11.2 PDSketch

We focus on the problem of learning models for a robot that operates in a space $\mathcal{S}$ of world states that plans to achieve goal conditions that are subsets of $\mathcal{S}$. A planning problem is a tuple $\langle \mathcal{S}, s_0, g, \mathcal{A}, \mathcal{T} \rangle$, where $s_0 \in \mathcal{S}$ is the initial state, g is a goal specification in first-order logic, $\mathcal{A}$ is a set of actions that the agent can execute, and $\mathcal{T}$ is a environmental transition model $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$. The task of planning is to output a sequence of actions $\bar{a} = \{a_i \in \mathcal{A}\}$ in which the terminal state s_T induced by applying a_i sequentially following $\mathcal{T}$ satisfies the goal specification g: $eval(g, s_T) = 1$. The function $eval(g, s)$ determines whether state s satisfies the goal condition g by recursively evaluating the logical expression and using learned neural groundings of the primitive terms in the expression.

At execution time, the agent will observe s_0 and be given g from human input, such as a first-order logic expression corresponding to "all the apples are in a blue bowl." However, we do not assume that the agent knows, in advance, the groundings of g (i.e. the underlying $eval(g, s)$ function) or the transition model $\mathcal{T}$. Thus, we need to learn g and $\mathcal{T}$ from data, in the form of observed trajectories that achieve goal states of interest.

Formally, we assume the training data given to the agent is a collection of tuples $\langle \overline{s}, \overline{a}, g, \overline{succ} \rangle$, where $\overline{s} = \langle s_i \rangle$ is a sequence of world states, $\overline{a} = \langle a_i \rangle$ is the sequence of actions taken by the robot, g is a goal specification, and $\overline{succ} = \langle succ_i \rangle$ is the "task-success" signal. Each $succ_i \in \{0, 1\}$ indicates whether the goal g is satisfied at state s_i: $succ_i = eval(g, s_i)$. The data sequences should be representative of the dynamics of the domain but need not be optimal goal-reaching trajectories.

It can be difficult to learn a transition model that is accurate over the long term on some types of state representations. For this reason, we generally assume an arbitrary latent space, Φ, for planning. The learning problem, then, is to find three parametric functions, collectively parameterized by θ: state encoder $\mathcal{E}_\theta : \mathcal{S} \rightarrow \Phi$, goal-evaluation function $eval_\theta : \mathcal{G} \times \Phi \rightarrow \{0, 1\}$, and transition model $\mathcal{T}_\theta : \Phi \times \mathcal{A} \rightarrow \Phi$. Although the domain might be mildly partially observable or stochastic, our goal will be to recover the most accurate possible deterministic model on the latent space.

Local, sparse structure. We need models that will generalize very broadly to scenarios with different numbers and types of objects in widely varying arrangements. To achieve this, we exploit structure to enable compositional generalization: throughout this work we will be committing to an object-centric representation for s, a logical language for goals g, and a sparse, local model of action effects.

We begin by factoring the environmental state s into a set of object states. Each $s \in \mathcal{S}$ is a tuple $(\mathcal{U}_s, f_s)$, where $\mathcal{U}_s$ is the set of objects in state s, denoted by arbitrary unique names

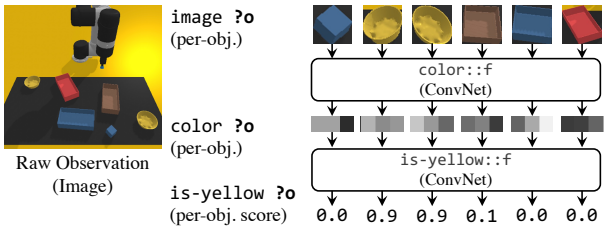
(e.g., `item#1`, `item#2`). The object set $\mathcal{U}_s$ is assumed to be constant within a single episode, but may differ in different episodes. The second component, f_s, is a dictionary mapping each object name to a fixed-dimensional object-state representation, such as a local image crop of the object and its position. We can extend this representation to relations among objects, for example by adding $g_s(x, y), x, y \in \mathcal{U}$ as a mapping from each object pair to a vector representation. We assume the detection and tracking of objects through time is done by external perception modules (e.g., object detectors and trackers).

We carry the object-centric representation through to actions, goals, and the transition model. Specifically, we define a *predicate* as a tuple $\langle name, args, grounding \rangle$, where *args* is a list of k arguments and *grounding* is a function from the latent representations of the objects corresponding to its arguments (in Ψ^k) into a scalar or vector value. (This is a generalization of the typical use of the term "predicate," which is better suited for use in robotics domains in which many quantities we must reason about are continuous.) For example, as illustrated in Fig. 11-3c, the predicate `wetness` takes a single argument as its input, and returns a (latent) vector representation of its wetness property; its grounding might be a neural network that maps from the visual appearance of the object to the latent wetness value. Given a set of predicates, we define the language of possible goal specifications to be all first-order logic formulas over the subset of the predicates whose output type is Boolean. To evaluate a goal specification in a state $(\mathcal{U}, f)$, quantification is interpreted in the finite domain $\mathcal{U}$ and f_s provides an interpretation of object names into representations that can serve as input to the grounded predicates.

The transition model $\mathcal{T}_\theta$ is specified in terms of a set object-parameterized action schemas $\langle name, args, precond, effect, \pi \rangle$, where *name* is a symbol, *args* is a list of symbols, *precond* and *effect* are descriptions of the action's effects, described in section 11.2.1, and π is a parameterized primitive policy for carrying out the action in terms of raw perception and motor commands. These local policies can be learned via demonstration or reinforcement learning in a phase prior to the model-learning phase, constructed using principles of control theory, or a combination of these methods. The set of concrete actions $\mathcal{A}$ available in a state s is formed by instantiating the action schema with objects in universe $\mathcal{U}_s$. We assume the transition dynamics of the domain (i.e., the *effect* of each action schema) are well characterized in terms of the changes of properties and relations of objects and that the transition model is lifted in the sense that it can be applied to domain instances with different numbers and types of objects. In addition, we assume the dynamics are local and sparse, in the sense that effects of any individual action depend on and change only a small number attributes and relations of a few objects, and that by default all other objects and attributes are unaffected. Taking again action schema `move-into` as an example, shown in Fig. 11-3d, only the states of object #1 and #6 are relevant to this action (but not #2, #3,

```
(:predicates ; input features
  (image [return_type=vector[float32], input] ?o - item)
)
(:derived
  (color [return_type=vector[float32]] ?o - item)
  (??f (image ?o))
)
(:derived
  (is-yellow ?o - item)  ; parameter and type
  (??f (color ?o))       ; function body
)
```

(a) A PDSketch definition of an input feature of each objects: `(image ?o)`, and two derived feature/predicates: `(color ?o)` and `(is-yellow ?o)`.



(b) The corresponding "computation graph" induced by the definitions. `color::f` and `is-yellow::f` are customizable CNNs that are applied identically to each object in the input state.

Figure 11-4: **A minimal example of defining derived features and predicates with blanks "??".**

etc.), and furthermore, the action only changes the `pose` and `wetness` properties of `item#1` (but not the color and the type).

The factored representation also introduces a factored learning problem: instead of learning a monolithic neural network for $\mathcal{T}_\theta$ and $eval_\theta$, the problem is factored into learning the grounding of individual predicates that appear in goal formulas, as well as the transition function for individual factors that were changed by an action.

11.2.1 Representation Language

The overall specification of $eval_\theta$ and $\mathcal{T}_\theta$ can be decomposed into two parts: 1) the locality and sparsity structures and 2) the actual model parameters, θ, such as neural network weights. We provide a symbolic language for human programmers to specify the locality and sparsity structure of the domain and methods for representing and learning θ. If the human provides no structure, the model falls back to a plain object-centric dynamics model [Zhu *et al.*, 2018]. However, we will show that explicit encoding of locality and sparsity structures can substantially improve the data efficiency of learning and the computational efficiency of planning with the resulting models.

PDSketch is an extension of the planning-domain definition language [Fikes and Nilsson, 1971; Fox and Long, 2003], a widely used formalism that focuses exposing locality and sparsity structure in symbolic planning domains. The key extensions are 1) allowing vector values in the computation graph and 2) enabling the programmer to use "blanks", which are unspecified functions that will be filled in with neural networks learned from data. Thus, rather than specifying the model in full detail, the programmer provides only a "sketch" [Solar-Lezama, 2008]. The two key representational components of PDDL are predicates and action schemas (operators).

Fig. 11-4 shows a simple example of PDSketch definition of predicates. All three predicates take a single object `?o` of type `item` as their argument, and return either a floating-

point vector or a scalar value from 0 to 1, indicating the score of a binary classifier. The `image` predicate simply refers to the raw image crop feature of the object. The `is-yellow` predicate's *grounding* takes a very simple form "`(??f (color ?o))`". The term `??f` defines a slot whose name is `f`. It takes only one argument, the `color` of the object `?o`, and outputs a classification score, which can be interpreted as the score of the object `?o` being yellow. The actual computation of the `yellow` predicate from the `color` value (as well as the computation of the `color` value from the `image` value) is instantiated in a neural network with trained parameters. The computation graph for the whole model can be built by recursively chaining the function bodies of predicate definitions.

Next, we illustrate how *locality* and *sparsity* structures can be specified for an action schema. Fig. 11-5 defines an action schema *name* `move-into` with two effect components. First, highlighted in blue, the action changes the pose of object `?o1` to a new pose that depends on the current pose of the second object `?o2`. Rather than hand-coding this detailed dependence, we leave the grounding blank. In addition, in our domain, the `wetness` of an object may be changed when the object is placed into a specific type of container. This is encoded by specifying a conditional effect using the keyword `when`, with two parts: 1) a Boolean-valued condition `g` of some other predicates on the state (in this case, the types of the two objects), and 2) the actual "effect", in this case, to change the `wetness` of `?o1` based on a function that considers the current `wetness` of `?o1`. The update will be applied only if the condition is true. To ensure the computation is differentiable, we make this condition "soft": let w be the current wetness, w' be the new wetness computed by function `??h`, and c be the scalar condition value computed by function `??g`. The updated value of the wetness will be $cw' + (1 - c)w$. Note that all the pose, wetness, and type representations can be arbitrary latent vectors computed by an encoder from the raw input. Thus, the "blanks" `??f`, `??g`, `??h` are indeed general neural networks. The effect definition here induces a corresponding computation graph of neural network weights and state representation tensors.

Like PDDL, PDSketch has full support of first-order logic, including Boolean operations (and, or, not) and finite-domain quantifiers ($\forall$ and $\exists$). They allow us to define more complex structures in the domain of interest. We present our full language and more examples in Appendix K.2.

11.2.2 Model Learning and Planning with PDSketch

Let θ denote the collection of all learnable parameters required to complete a PDSketch domain definition into a full model. This includes the parameters of the state encoder, all predicate groundings, and the definitions of slot-update functions that were left blank in the sketch. Recall that our training data are tuples of three sequences and a goal formula $\langle \overline{s}, \overline{a}, g, \overline{succ} \rangle$. Fundamentally, our objective is to minimize a sum of two losses, one related

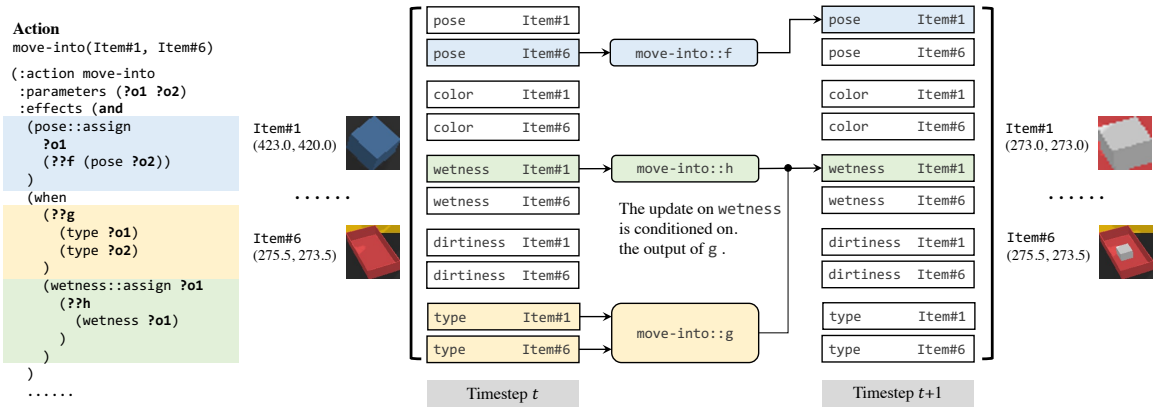


Figure 11-5: **A computation graph of a partial definition of the action "move-into."**

to predicting the truth values of the goal formulas and one related to predicting the next state, given the previous state and action:

$$\mathcal{L}(\theta) = \sum_{\langle \bar{s}, \bar{a}, g, \overline{succ} \rangle \in \mathcal{D}} \left\{ \sum_i \text{BCE}\left(eval_\theta(g, \mathcal{E}_\theta(s_i)), succ_i\right) + \sum_i \text{L1}\left(\mathcal{T}_\theta(\mathcal{E}_\theta(s_i), a_i), \mathcal{E}_\theta(s_{i+1})\right) \right\},$$

where BCE is the binary cross-entropy classification loss and L1 is a regression loss. To avoid degenerate local optima, we add a "lookahead" loss term that combines both aspects, as detailed in the Appendix K.1.2. If the encoder $\mathcal{E}_\theta$ is constrained to be the identity, and there is no predicate-level structure, then the transition model essentially learns to be a per-object next-image predictor. More generally, including the encoder turns this into a bisimulation objective [Li *et al.*, 2006]: we want to uncover a latent transition model that accurately predicts the reward signal (in this case, whether the goal is satisfied or not) but does not necessarily reconstruct the input state representation.

In order to adjust θ to minimize this loss, we must establish a differentiable computation graph. The encoder $\mathcal{E}_\theta$ will generally be a relatively standard combination of convolutional and fully-connected feed-forward neural network. Fig. 11-5 illustrates the computation graph associated with $\mathcal{T}_\theta$ for a particular choice of action a_i and objects that serve as its arguments. Importantly, note that there is a substantial amount of parameter-tying: the same predicate-grounding network might appear in multiple times, even when characterizing $\mathcal{T}_\theta$ for a single action, if that predicate appears multiple times (applied to different objects) in the preconditions or effects of the action. The computation graph shown here, as well as those necessary to compute $eval_\theta(g, \mathcal{E}_\theta(s_i))$, involves Boolean operators, which do not have useful derivatives for optimization. We address this by representing truth values as elements of the interval $[0, 1]$ and approximate logical operations with the differentiable Gödel t-norms: $\texttt{not}(p) = 1 - p$, $\texttt{and}(p_1, p_2) = \min(p_1, p_2)$, $\forall x.p(x) = \min_x p(x)$.

Once we have estimated θ from data, we can solve any planning problem in the domain given any starting state $s_0 \in \mathcal{S}$ and goal g expressed in terms of predicates for which we have groundings. The resulting transition model $\mathcal{T}$ can be used by a variety of different planners. We will focus on forward search, constructing a tree rooted at $\mathcal{E}(s_0)$ with branches corresponding to the possible instantiations, a, of action templates $\mathcal{A}$ with the objects in the universe $\mathcal{U}$ associated with s_0, and next latent states computed by applying $\mathcal{T}$. The search terminates when it reaches a node n in which $\text{eval}_\theta(n, g) > .5$. Unguided forward search can be very slow when the planning horizon is long or the branching factor is large (e.g., when there are many objects in an environment). To address this, we will use the A^* heuristic search algorithm using a *domain-independent search heuristic* directly derivable from the locality and sparsity structure defined in PDSketch.

11.2.3 Inducing Domain-Independent Heuristics

Because of the rich representational capacity of PDSketch models, in which values can be continuous and multidimensional, we cannot take advantage of the planning algorithms that operate on PDDL input, such as Fast Downward [Helmert, 2006], which derive much of their efficiency from domain-independent heuristics. We cannot use their strategies in detail, but we take inspiration from the idea of deriving an optimistic estimate of the cost to reach the goal from a state by solving a "relaxed" version of the problem, which is computationally easier than the original [Bonet and Geffner, 2001].

One way to construct a relaxed planning problem is to allow each predicate instance to take on multiple values at the same time. For example, the robot or an object can effectively be in multiple places at the same time. In this relaxation, computing the number of steps needed to change the value of a predicate instance can be done in polynomial time. We can use such a relaxation in the hFF heuristic [Hoffmann and Nebel, 2001] to get an estimate of the cost to goal, by first chaining the actions forward until all the components of the goal condition have been made true, and then searching to recover a small set of actions that can collectively achieve the goal under the relaxation.

To use hFF, however, we must reduce our continuous-space problem to a discrete-space one. Specifically, we discretizes all continuous state variables (poses, etc.) into a designated number of bins (e.g., 128). Next, for each externally-defined functions, we learn a first-order decision tree to approximate the computation (e.g., to approximate the neural network). Concretely, we use VQVAE [Van Den Oord *et al.*, 2017] to discretize the feature vectors. For all predicates whose output is a latent embedding, we add a vector quantization layer after the encoding layers. We initialize the quantized embeddings by running a K-means clustering over the item feature embedding from a small dataset, and finetune the weights on the entire training dataset for one epoch. We describe this in more detail in Appendix K.3.

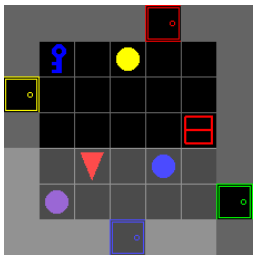


Figure 11-6: A screenshot of the BabyAI environment.

Model	Inductive Biases				Succ. Rate	
	Obj-Centric.	Facing	Rob. Dyn.	Obj. Prop.	Basic	#Obj.Gen.
BC	Y	N	N	N	0.93	0.79
DT (S)	Y	N	N	N	0.91	0.82
DT (S+F)	Y	N	N	N	0.32	0.19
DreamerV2	N	N	N	N	0.96	0.79
PDS-Base	Y	N	N	N	0.82	0.62
PDS-Abs	Y	Y	N	N	0.99	0.98
PDS-Rob	Y	Y	Y	N	**1.00**	**1.00**

Table 11.1: The planning success rate of different models on BabyAI.

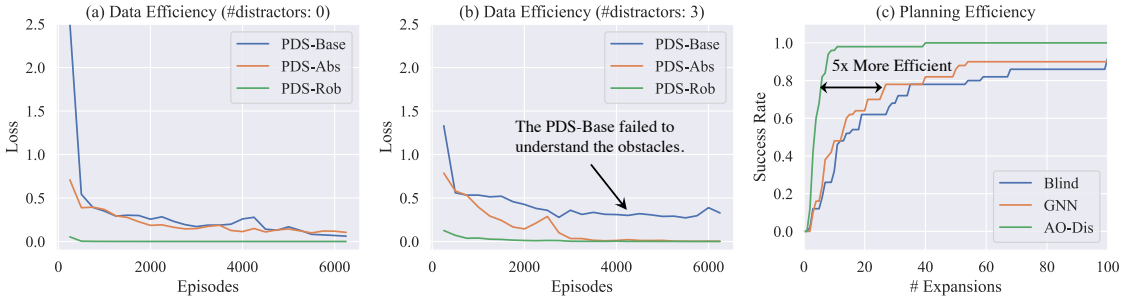


Figure 11-7: (a) and (b) Data efficiency comparison for model learning. We compare three structures with different levels of abstractness. (c) Planning efficiency, measured as the number of expanded nodes for different heuristic computation methods.

Although our discretizations are inherently lossy on non-Boolean predicate values, since they are only used for search guidance, and not in the forward simulation of the actual actions during planning, this does not affect the correctness of the overall algorithm.

11.3 Experiments

We evaluate PDSketch in two domains: BabyAI, a 2D grid-world and Painting Factory, a simulated tabletop manipulation task. We compare our model with two model-free methods: Behavior Cloning [BC; Bain and Sammut, 1995] and Decision Transformer [DT; Chen *et al.*, 2021b]. We implement two DT variants: DT-S with only successful demonstrations, and DT-S+F with successful and failed demonstrations. We use graph neural networks [Gori *et al.*, 2005; Battaglia *et al.*, 2018] as their state encoder.

```
(:action forward-abs
 :parameters (?r - robot)
 :precondition (and )
 :effect (robot-pose::cond-assign ?r
   (??f
     (robot-pose ?r) (robot-dir ?r)
     (foreach (?o - item)
       (item-feature::cond-select ?o (facing ?r ?o))
     )
   )
   (??g (robot-pose ?r) (robot-dir ?r))
))
```

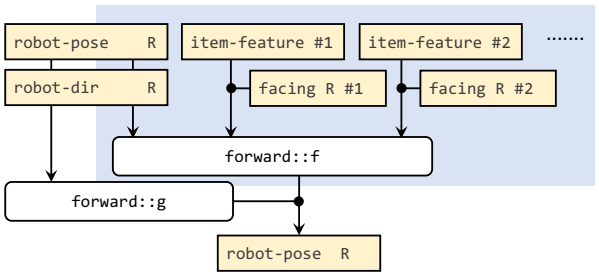


Figure 11-8: **The PDSketch definition of the "forward" action and the corresponding computation graph for the PDS-Abs model.** The mentioned predicate `facing` is an derived predicate represented as a neural network that is jointly learned (omitted in the graph for brevity).

11.3.1 BabyAI

BabyAI [Chevalier-Boisvert *et al.*, 2019] is an image-based 2D grid-world environment where an agent can navigate around obstacles, pick and place objects, and toggle doors. In this chapter, we focus on a specific level of BabyAI, namely *ActionObjDoor*. At this level, the agent navigates within a 7x7 grid. The goals include *go to an X*, *pick up an X*, *open an X*, where *X* is a noun phrase, such as "blue key". We train all models on environments with 4 doors and 4 objects. The offline dataset $\mathcal{D}$ contains both successful and failure demonstrations obtained by A^* search. We extend PDSketch to interactive data gathering in Appendix K.1.2. Additionally, we test generalization to environments with 6 doors and 8 objects. Since objects may block agents, the agent needs to successfully uncover the underlying dynamics and plan to navigate around them.

We study three PDSketch models with various levels of *locality* and *sparsity* structures built in. The PDS-Base model has no built-in structures: each object is represented as a holistic vector. This falls back to an object-centric transition model [Zhu *et al.*, 2018]. In contrast, the PDS-Abs model disentangles poses from object appearances. Importantly, it defines a predicate *facing* and uses this concept to write action definitions (e.g., whether a robot move will be blocked by the object it is facing). However, the grounding of *facing* is to be learned. Figure 11-8 shows the detail definition of the forward action in the PDS-Abs model. PDS-Rob contains predefined rules for robot movements but the object recognition modules are learned. We provide their definitions in Appendix K.4.

Results. Table 11.1 shows the results. Overall, PDSketch with more structure (PDS-Abs and PDS-Rob) outperforms baselines by a significant margin. From the performance of PDS-Abs, we see that even a tiny amount of additional structure (e.g., an *ungrounded* predicate "facing") significantly improves the performance, especially when it comes to generalization to more complex environments. See below for a zoomed-in analysis for

model learning. Furthermore, we hypothesize that the inferior performance of decision transformer in the mixed training data (Succ+Fail) setting is due to the reward sparsity: the agent only gets reward 1 when it reaches the goal. Thus, the failed demonstrations are generally hard to model as they are irrelevant to the goal specification. In addition, we compare our models with DreamerV2, a state-of-the-art model-based reinforcement learning algorithm for image-based environments. Compared with BC and DT, we see DreamerV2 achieves slightly improved performance for the basic task, but does not show stronger generalization to environments with more objects. We hypothesize this is because Dreamer still learns a fixed policy for execution.

Data efficiency. Fig. 11-7a-b shows model learning curves for the three PDSketch models, in the cases where there is a single object in the environment and when there are 4 objects. . When the number of object is small, the environmental dynamics is easier to learn: both Base and Abs have a similar performance. However, when the number of object increases, PDSketch can leverage the inductive bias to learn the dynamics faster. Shown in the figure, even at the end of training, the model PDS-Base has not successfully learned the correct movement dynamics, leading to its inferior performance during generalization to more objects.

Planning runtime efficiency. Fig. 11-7c quantifies the number of nodes expanded by A^* when using different heuristic computations. Specifically, we compare our model with the *blind* heuristic (i.e., the heuristic value of a state is 0 when it satisfies the goal and otherwise 1.) and a GNN-parameterized heuristic learned from successful demonstrations. All results are based on the PDS-Abs model. First, learning-based heuristics (GNN) shows improvement over the baseline "blind" heuristic, especially when the task is easy (requiring a small number of expansions). Second, the heuristic derived from PDSketch significantly improves the search efficiency. At the success rate of 0.8, our method is 5 times more efficient than learning-based heuristics.

11.3.2 Robot Painting

Finally, we extend the framework to a tabletop manipulation task, built based on the tabletop environment of Zeng *et al.* [2020a]. There are three zones, several bowls, and several blocks on the table. The robot can use its suction gripper to pick-and-place objects into a designated location. Blocks have 8 possible colors. Placing objects in a bowl will paint the object to be the same color as the bowl. The task is to paint the blocks and organize them in the target brown box. Our training-time goal requires the robot to paint-and-place two objects. The goal contains their colors and their relationship (e.g., a *pink* block is *left of* a *yellow* block.

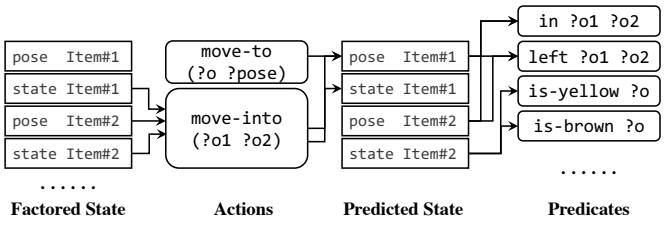


Figure 11-9: Our encoding of the painting factory domain. Each object has a pose and a state. Action `move-to` directly changes the object pose. `move-into` moves an object into a container, which possible changes the state (e.g., color) of the object. Goals are represented using relational predicates.

Model	Succ. Rate @100	Succ Rate @1000
BC	0.70	0.95
DT	0.56	0.97
PDS	**0.91**	**0.99**

Table 11.2: Model performance on the painting factory domain, measured as plan success rate. The number after @ is the number of demonstration trajectories. PDSketch models show stronger data efficiency than baselines.

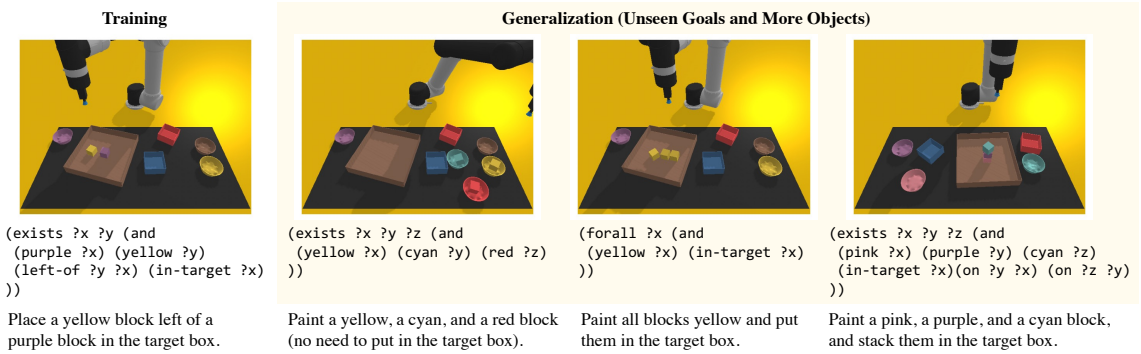


Figure 11-10: **Generalization results on the painting factory domain.** From only demonstrations of one training task family (left: Place an X and a Y into the target box), a model specified in PDSketch learns primitive concepts including object colors and spatial relations. They can be used in planning for unseen tasks given new first-order logic descriptions of the goal. The natural language descriptions are shown for readability.

Demonstration are collected using hand-crafted oracle policies following Zeng *et al.* [2020a]. The offline dataset contains only successful trajectories. There are two built-in actions (move-to, and move-into) that the robot can execute. Fig. 11-9 shows the computation graph derived from PDSketch.

State and action space. The state space in the painting factory is composed of a list of objects, and a list of containers. Both objects and containers are represented as a tuple of a 3D xyz location, and a image crop. The image crop is generated by first computing the 2D bounding box of the object in the camera plane, cropping out the image patch, and resizing it to 32 by 32. The action space contains two primitives. First, `move-into` is an action defined for each pair of item and container. It changes the pose of the item (now the item will be inside the container). When the item is placed into a bowl, it will be painted into the same color as the bowl. This action simplifies the action generation procedure since it does not require the model to output a "placement" pose of the object. The second action takes an object and a 3D location as input, and moves the object directly to a designated position (ignoring the rotation).

Results. Table 11.2 shows the planning success rate *on the training task* with different amounts of training data. Overall, PDSketch is more data efficient than both baselines and achieves strong overall performance in this task. Much more importantly, Fig. 11-10 shows our *generalization to novel task specifications*, which involve more objects or new specifiers (e.g., `forall`). Our model generalizes directly to these novel scenarios *without any additional training*. The quantitative performance for these three generalization tasks are: 0.99, 0.98, and 0.87, respectively, measured as success rate after executing the plan. The last task has lower success rate because stacking objects may fail due to controller and physical noises. We specify implementation details in Appendix K.4.

11.4 Behavior from Language and Demonstration

While PDSketch assumes domain structures are specified by humans, many real-world settings demand automatic discovery of these structures. This motivates BLADE (Behavior from LAnguage and Demonstration). Instead of relying on hand-specified structures, BLADE leverages language-annotated demonstrations and large language models to acquire domain abstractions automatically. By segmenting demonstrations into contact-based primitives, querying LLMs for precondition–effect structures, and grounding them in perception, BLADE builds behavior representations that support long-horizon planning.

In this section, we tackle the problem of automatic acquisition of an abstraction for the

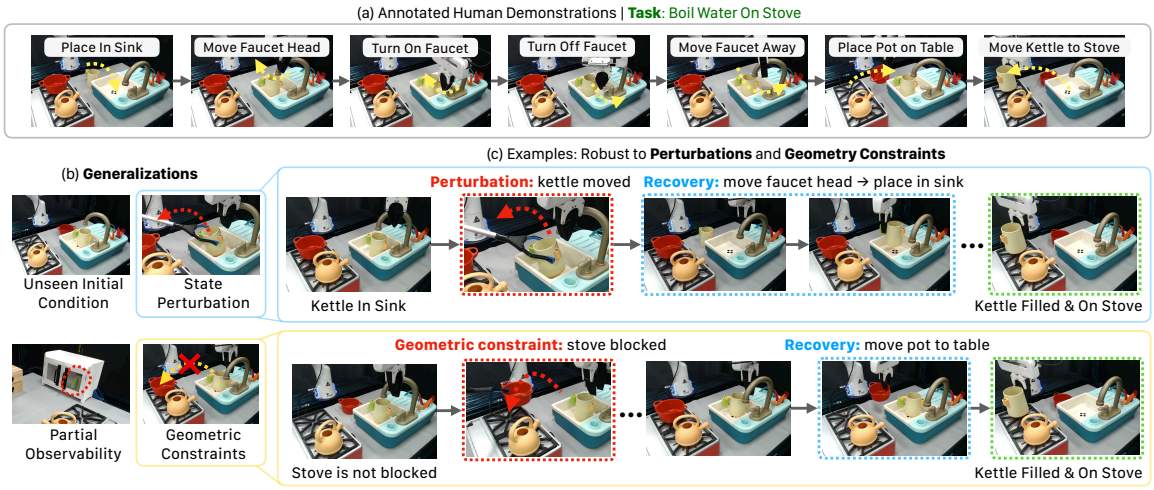


Figure 11-11: **Behavior from language and demonstration.** BLADE is a robot manipulation framework combining imitation learning and model-based planning. (a) BLADE takes language-annotated demonstrations as training data. (b) It generalizes to unseen initial conditions, state perturbations, and geometric constraints. (c) In the depicted scenarios, BLADE recovers from perturbations such as moving the kettle out of the sink, and resolves geometric constraints including a blocked stove.

learned "behaviors" to support long-horizon planning. The goal of this behavior abstraction learning is to build representations that describe the preconditions and effects of behaviors, to enable chaining and search. These representations should depend on the environment, the set of possible goals, and the specifications of individual behaviors. Furthermore, these representations should be grounded on high-dimensional perception inputs and low-level robot control commands.

Our insight into tackling this challenge is to leverage knowledge from two sources: the low-level, mechanical understanding of robot-object contact (Chapter 10), and the high-level, abstract understanding of object-object interactions described in language (Section 11.2.1) that can be extracted from language models as the knowledge source. Our framework, behavior from language and demonstration (BLADE), takes as input a small number of language-annotated demonstrations (Fig. 11-11a). It segments each trajectory based on which object is in contact with the robot. Then, it uses a large language model (LLM), conditioned on the contact sequences and the language annotations, to propose abstract behavior descriptions with preconditions and effects that best explain the demonstration trajectories. During training, we extract the state abstraction terms from the preconditions and effects (e.g., *turned-on*, *aligned-with*), and learn their groundings on perception inputs. We also learn the control policies associated with each behavior (e.g., *turn on the faucet*).

Our model offers several advantages. First, unlike prior work that relies on manually defined state abstractions or additional state labels, our method automatically generates

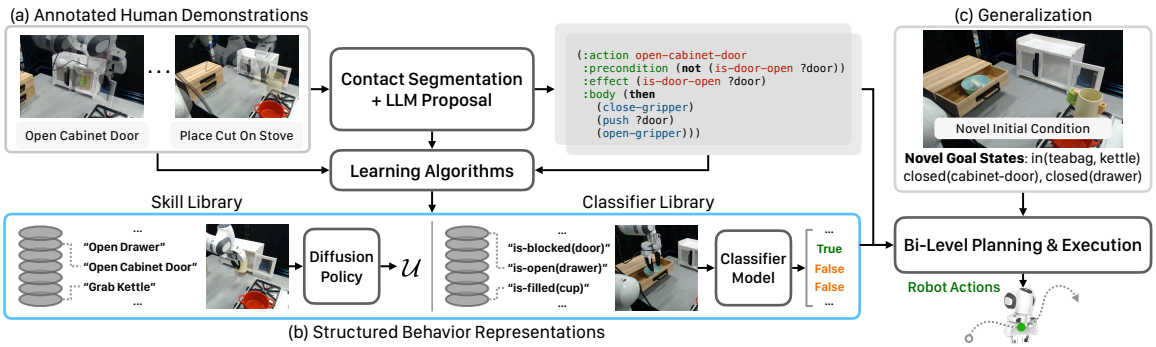


Figure 11-12: **Overview of BLADE.** (a) BLADE receives language-annotated human demonstrations, (b) segments demonstrations into contact primitives, and learns a structured behavior representation. (c) It generalizes to novel conditions by leveraging bi-level planning and execution to achieve goal states.

state abstraction labels based on the language-annotated demonstrations and LLM-proposed behavior descriptions. BLADE recovers the visual grounding of these abstractions without any additional labels. Second, BLADE generalizes to novel states and goals by composing learned behaviors using a planner. Shown in Fig. 11-11b, it can handle various novel initial conditions and external perturbations that lead to unseen states. Third, our method can handle novel geometric constraints (Fig. 11-11c) and partial observability from articulated bodies like drawers.

11.4.1 Problem Formulation

Concretely, we consider the problem of learning a language-conditioned goal-reaching manipulation policy. Formally, the environment is modeled as a tuple $\langle \mathcal{X}, \mathcal{U}, \mathcal{T} \rangle$ where $\mathcal{X}$ is the raw state space, $\mathcal{U}$ is the low-level action space, and $\mathcal{T} : \mathcal{X} \times \mathcal{U} \to \mathcal{X}$ is the transition function (which may be stochastic and unknown). Furthermore, the robot will receive observations $o \in \mathcal{O}$ that may be partially observable views of the states. At performance time, the robot also receives a natural language instruction ℓ_t, which corresponds to a set of goal states. An oracle goal satisfaction function defines whether the language goal is reached, i.e., $g_{\ell_t} : \mathcal{X} \to \{T, F\}$, but the model does not have access to it. Given an initial state $x_0 \in \mathcal{X}$ and the instruction ℓ_t, the robot should generate a sequence of low-level actions $\{u_1, u_2, ..., u_H\} \in \mathcal{U}^H$.

In the language-annotated learning setting, the robot has a dataset of language-annotated demonstrations $\mathcal{D}$. Each demonstration is a sequence of robot actions $\{u_1, ..., u_H\}$ paired with observations $\{o_0, ..., o_H\}$. Each trajectory is segmented into M sub-trajectories, and natural language descriptions $\{\ell_1, ..., \ell_M\}$ are associated with the segments (e.g., "*place the kettle on the stove*"). In this section, we assume that there is a finite number of possible ℓ's—each

corresponding to a skill to learn.

Directly learning a single goal-conditioned policy that can generalize to novel states and goals is challenging. Therefore, we recover an *abstract* state and action representation of the environment and combine online planning in abstract states and offline policy learning for low-level control to solve the task. In BLADE, behaviors are represented as temporally extended actions with preconditions and effects characterized by state predicates. Formally, we want to recover a set of predicates $\mathcal{P}$ that define an abstract state space $\mathcal{S}$. We focus on a scenario where all predicates are Boolean-valued. However, they are grounded on high-dimensional sensory inputs. Using $\mathcal{P}$, a state can be described as a set of grounded atoms such as $\{\mathit{kettle}(\mathrm{A}), \mathit{stove}(\mathrm{B}), \mathit{filled}(\mathrm{A}), \mathit{on}(\mathrm{A}, \mathrm{B})\}$ for a two-object scene. BLADE will learn a function $\Phi : \mathcal{O} \to \mathcal{S}$ that maps observations to abstract states. In its current implementation, BLADE requires humans to additionally provide a list of predicate names in natural language, which we have found to be helpful for LLMs to generate action definitions. We provide additional ablations in the Appendix L.1.2. Based on $\mathcal{S}$, we learn a library of *behaviors* (a.k.a., *abstract actions*). Each behavior $a \in \mathcal{A}$ is a tuple of $\langle \mathit{name}, \mathit{args}, \mathit{pre}, \mathit{eff}, \pi \rangle$: *name* is the name of the action. *args* is a list of variables related to the action, often denoted by $?x, ?y$; *pre* and *eff* are the precondition and effect formula defined in terms of the variables *args* and the predicates $\mathcal{P}$. A low-level policy $\pi : \mathcal{O} \to \mathcal{U}$ is also associated with a. The semantics of the preconditions and effects is: for any state x such that $\mathit{pre}(\Phi(x))$ is satisfied, executing π at x will lead to a state x' such that $\mathit{eff}(\Phi(x'))$ [Lifschitz, 1987].

BLADE is a method for learning abstract state and action representations from language-annotated demonstrations. It works in three steps, as illustrated in Fig. 11-12. First, we generate a symbolic behavior definition conditioned on the language annotations and contact sequences in the demonstration using a large language model (LLM). Next, we learn the classifiers associated with all state predicates and the control policies, all from the demonstration without additional annotations. At test time, we use a combined planning and execution strategy to generate robot actions.

11.4.2 Behavior Description Learning

Given a finite set of behaviors with language descriptions $\{\ell\}$ and corresponding demonstration segments, we generate an abstract description for each ℓ by querying large language models. To facilitate LLM generation, we provide additional information on the list of objects with which the robot has contact. The generated operators are further refined with abstract verification.

Temporal segmentation. We first segment each demonstration (Fig. 11-13a) into a sequence of *contact-based primitives* (Fig. 11-13b). In this section we consider seven primi-

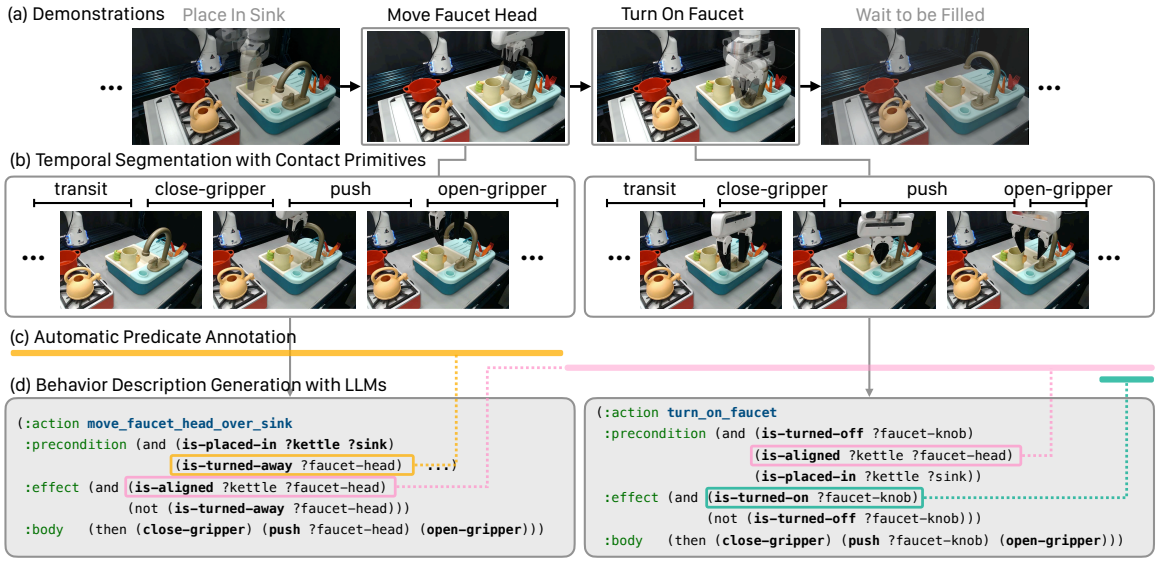


Figure 11-13: **Behavior descriptions learning in BLADE.** (a) A demonstration is provided along with corresponding language annotations. (b) The demonstration is segmented into a sequence of contact primitives. (c) A large language model interprets the annotation and contact sequence, generating a symbolic behavior definition. (d) The system automatically generates data to learn classifiers for state predicates.

tives describing the interactions between the robot and other objects: *open*/*close* grippers without holding objects, *move-to*(x) which moves the gripper to an object, *grasp*(x, y) and *place*(x, y) which grasp and place object x from/onto another object y, *move*(x) which moves the currently holding object x and *push*(x). We leverage proprioception, i.e., gripper open state, and object segmentation to automatically segment the continuous trajectories into these basis segments. For example, pushing the faucet head away involves the sequence of {*close-gripper*, *push*, *open-gripper*}. This segmentation will be used for LLMs to generate operator definitions and for constructing training data for control policies.

Behavior description generation with LLMs. Our behavior description language is based on PDDL [Fox and Long, 2003]. We extend the PDDL definition to include a *body* section, which is a sequence of contact primitives as in Chapter 10. It will be generated by the LLM based on the demonstration data.

Our input to the LLM mainly contains: 1) a general description of the environment, 2) the natural language descriptions ℓ associated with the behavior itself and other behaviors that have appeared preceding or following ℓ in the dataset, 3) all possible sequences of contact primitive sequences associated with ℓ across the dataset, and 4) additional instructions on the PDDL syntax, including a single PDDL definition example. We find the additional context useful. As shown in Fig. 11-13d, in addition to preconditions and effects of the operators,

we also ask LLMs to predict the *body* as a sequence of contact primitives associated with the behavior, which we call *body*. We assume that each behavior has a single corresponding contact primitive sequence, and use this step to account for noise in the segmentation annotations. After the LLM predicts the definition for all behaviors, we will re-segment the demonstrations associated with each behavior based on the LLM-predicted body section.

Behavior description refinement with abstract verification. In addition to checking for syntax errors, we also verify the generated behavior descriptions with *abstract verification* on the demonstration trajectories. Given a segmented sequence of the trajectory where each segment is associated with a behavior, we verify whether the preconditions of each behavior can be satisfied by the accumulated effects of the previous segments. This verification does not require learning the grounding of predicates and can be done at the behavior level for incorrect preconditions and effects, and at the contact primitive level for missing or incorrect contact primitives (e.g., *grasp* cannot be immediately followed by other *grasp*). We resample behavior definitions that do not pass the verification.

11.4.3 Classifier and Policy Learning

Given the dataset of state-action segments associated with each behavior, we train the classifiers for different state predicates and the low-level controller for each behavior.

Automatic predicate annotation. We leverage *all* behavior descriptions to automatically label an observation sequence $\{o_1, ..., o_H\}$ based on its associated segmentation. In particular, at o_0, we label all state predicates as "unknown." Next, we unroll the sequence of behavior executed in $\bar{o}$. As illustrated in Fig. 11-13c, before applying a behavior a at step o_t, we label all predicates in pre_a true and predicates in eff_a false. When a finishes at step $o_{t'}$, we label all predicates in eff_a. In addition, we will propagate the labels for state predicates to later time steps until they are explicitly altered by another behavior a. In contrast to earlier methods, such as Migimatsu and Bohg [2022] and Section 11.2.1, which directly use the first and last state of state-action segments to train predicate classifiers, our method greatly increases the diversity of training data. After this step, for each predicate $p \in \mathcal{P}$, we obtain a dataset of paired observations o and the predicate value of p at the corresponding time step.

Classifier learning. Based on the state predicate dataset generated from behavior definitions, we train a set of state classifiers $f_\theta(p) : \mathcal{O} \rightarrow \{T, F\}$, which are implemented as standard neural networks for classification. We include implementation details in Appendix L.1.6. In real-world environments with strong data-efficiency requirements, we additionally use an open vocabulary object detector [Liu *et al.*, 2024a] to detect relevant objects for the

state predicate and crop the observation images. For example, only pixels associated with the object faucet will be the input to the *turned-on*(faucet) classifier.

Policy learning. For each behavior, we also train control policies $\pi_\theta(a) : \mathcal{O} \to \mathcal{U}$, implemented as a diffusion policy [Chi *et al.*, 2023]. In simulation, we use a combination of frame-mounted and wrist-mounted RGB-D cameras as the inputs to the diffusion policy, while in the real world, the policy takes raw camera images as input. The high-level planner orchestrates which of these low-level policies to deploy based on the scene and states. Once trained on these diverse demonstrations of different skills, the resulting low-level policies can adapt to local changes, such as variations in object poses.

11.4.4 Planning and Execution

At test time, given a novel state and a novel goal, BLADE first uses LLMs to translate the goal into a first-order logic formula based on the state predicates. Next, it leverages the learned state abstractions to perform planning in a symbolic space to produce a sequence of behaviors. Then, we execute the low-level policy associated with the first behavior, and we re-run the planner after the low-level policy finishes—this enables us to handle various types of uncertainties and perturbations, including execution failure, partial observability, and human perturbation. In our implementation, we use the fast-forward heuristic to generate plans [Hoffmann and Nebel, 2001]; however, our method is planner-agnostic, and other symbolic planners (e.g., Fast-Downward [Helmert, 2006]) are compatible.

Visibility and geometric constraints are also modeled as preconditions, in addition to other object-state and relational conditions. For example, the behavior "opening the cabinet door" will have preconditions on the initial state of the door, a visibility constraint that the door is visible, and a geometric constraint that nothing is blocking the door. When those preconditions are not satisfied, the planner will automatically generate plans, such as actions that move obstacles away, to achieve them. Partial observability was handled by using the most-likely state assumption during planning and performing replanning. We include details in Appendix L.1.8.

11.4.5 Simulation Experiments

Setup. We use the CALVIN benchmark [Mees *et al.*, 2021] for simulation-based evaluations, which include teleoperated human-play data. We use the split D of the dataset, which consists of approximately 6 hours of interactions. Annotations of the play data are generated by a script that detects goal conditions on simulator states, and there are in total 34 types of behaviors. We use RGB-D images from the mounted camera for classifier

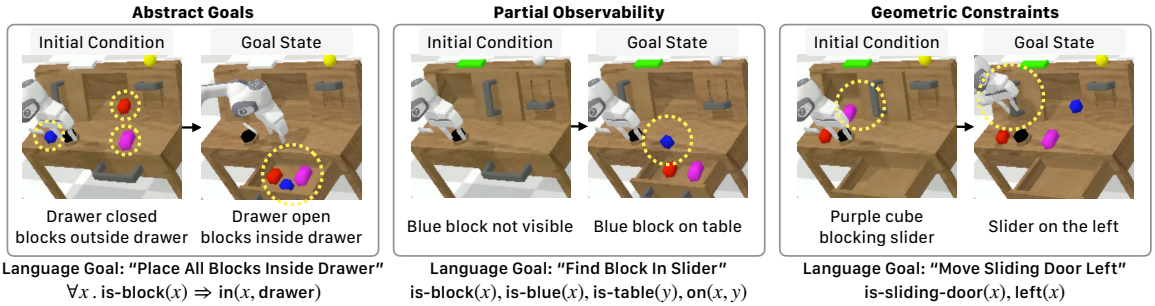


Figure 11-14: **Generalization tasks in CALVIN.** Examples from the three generalization tasks in the CALVIN simulation environment. Successfully completing these tasks require planning for and executing 3-7 actions.

Method	State Classifier	Latent Feasibility	Generalization Task		
			Abstract Goal	Geometric Constraint	Partial Observability
HULC [Mees *et al.*, 2022]	N/A	N/A	2.78 ± 3.47	11.67 ± 11.55	0.00 ± 0.00
SayCan [Ahn *et al.*, 2022]	N/A	Short	23.89 ± 1.92	1.67 ± 2.89	1.67 ± 2.89
VILA [Hu *et al.*, 2023]	N/A	N/A	18.38 ± 2.48	0.00 ± 0.00	4.17 ± 5.20
T2M-Shooting [Lin *et al.*, 2023]	Learned	Long	57.78 ± 12.29	0.00 ± 0.00	13.33 ± 1.44
Ours	Learned	N/A	**68.33 ± 10.14**	**26.67 ± 7.64**	**75.83 ± 3.82**
T2M-Shooting [Lin *et al.*, 2023]	GT	Long	61.67 ± 5.00	0.00 ± 0.00	0.83 ± 1.44
Ours	GT	N/A	**76.11 ± 6.74**	**56.67 ± 16.07**	**70.00 ± 5.00**

Table 11.3: **Generalization results on CALVIN.** Mean success rates with STD from three seeds are reported. BLADE outperforms latent planning, LLM, and VLM baselines in completing novel long-horizon tasks.

learning and partial 3D point clouds recovered from the images for policy learning. The original benchmark focuses only on evaluating individual skills and instruction following. To evaluate the ability to compose previously learned policies to solve novel tasks, we design six new generalization tasks, with examples shown in Fig. 11-14. Each task has a language instruction, a sampler that generates random initial states, and a goal satisfaction function for evaluation. For each task, we sample 20 initial states and evaluate all methods with three different random seeds. See Appendix L.2.1 for more details on the benchmark setup.

Baselines. We compare BLADE with two groups of baselines: hierarchical policies with planning in latent spaces and LLM/VLM-based methods for robotic planning. For the former, we use HULC [Mees *et al.*, 2022], a representative method in CALVIN, which learns a hierarchical policy from language-annotated play data using hindsight labeling. For the latter, we use SayCan [Ahn *et al.*, 2022], Robot-VILA [Hu *et al.*, 2023], and Text2Motion [Lin *et al.*, 2023]. Note that Text2Motion assumes access to ground-truth symbolic states. Hence we compare Text2Motion with BLADE in two settings: one with the ground-truth states and the other with the state classifiers learned by BLADE. See Appendix

L.2.2 for more details on these methods.

Table 11.3 presents the performance of different models in all three types of generalization tasks.

Structured behavior representations improve long-horizon planning. We first compare to the hierarchical policy HULC in Table. 11.3. BLADE with learned classifiers achieves a more than 65% improvement in the success rate for reaching abstract goals while using the same language-annotated play data. We attribute this to the particular implementation of hindsight labeling in HULC being not sufficient to generate plans that chain multiple high-level actions: for example, the task of placing all blocks in the closed drawer requires chaining together a minimum of 7 behaviors.

Structured transition models learned by BLADE facilitate long-horizon planning. Both SayCan and T2M-Shooting uses learned action feasibility models for planning. Shown in Table. 11.3, learning accurate feasibility models directly from raw demonstration data remains a significant challenge. In our experiment, we find that first, when the LLM does not take into account state information (SayCan), using the short-horizon feasibility model is not sufficient to produce sound plans. Second, since our model learns a structured transition model, factorized into different state predicates, BLADE is capable of producing more accurate longer-horizon plans than T2M-Shooting which learns long-horizon feasibility from data.

Structured scene representations facilitate making feasible plans. Compared to the Robot-VILA method, which directly predicts action sequences based on the image state, BLADE first uses learned state classifiers to construct an abstract state representation. This contributes to a 49% improvement on the Abstract Goal tasks in Table 11.3. We observe that the pre-trained VLM used in Robot-VILA often predicts actions that are not feasible in the current state. For example, Robot-VILA consistently performs better in completing "placing all blocks in a closed drawer" than "placing all blocks in an open drawer" since it always predicts opening the drawer as the first step.

Explicit modeling of geometric constraints and object visibility improves performance in these scenarios. BLADE can reason about these challenging situations without explicitly being trained in those settings. Table. 11.3 shows that our approach consistently outperforms baselines in these two settings. These generalization capabilities are built on the explicit modeling of geometric constraints and object visibility in behavior preconditions.

BLADE can automatically propose operators for the specific environment given demonstrations. Our experiment shows that the LLM can automatically propose high-quality

Method	Abstract	Geometric	Partial Obs.
Migimatsu and Bohg [2022]	33.89 ± 5.85	9.17 ± 5.20	3.33 ± 2.89
BLADE	**68.33 ± 10.14**	**26.67 ± 7.64**	**75.83 ± 3.82**

Table 11.4: **Ablation on state classifier learning on CALVIN.**

behavior descriptions that resemble the dependency structures among operators. For example, the LLM discovers from the given contact primitive sequences and language-paired demonstration that blocks can only be placed after the block is lifted and that a drawer needs to be opened before placing objects inside, etc. Some of these dependencies are unique to the CALVIN environment, therefore requiring the LLM to generate specifically for this domain. We provide more visualizations in the Appendix L.1.1.

BLADE's automatic predicate annotation enables better classifier learning. From Table 11.3, we observe that having accurate state classifier models is critical for algorithms' performance (GT vs. Learned). Hence, we perform additional ablation studies on classifier learning. Prior work by Migimatsu and Bohg [2022] also presented a method for learning the preconditions and effects of actions from segmented trajectories and symbolic action descriptions. The key difference between BLADE and their method is that they only use the first and last frame of each segment to supervise the learning of state classifiers. We compare the two classifier learning algorithms, given the same LLM-generated behavior definitions, by evaluating the classifier accuracy on held-out states. BLADE shows a 20.7% improvement in F1 (16.3% improvement for classifying object states and 38.6% improvement for classifying spatial relations) compared to the baseline model. This also translates into significant improvements in the planning success rate, as shown in Table 11.4.

11.4.6 Real-World Experiments

Environments. We use a Franka Emika robot arm with a parallel jaw gripper. The setup includes five RealSense RGB-D cameras, with one being wrist-mounted on the robot and the remaining positioned around the workspace. Fig. 11-15 shows the two domains: Make Tea and Boil Water. For each domain, we collect 85 language-annotated demonstrations using teleoperation with a 3D mouse. After segmenting the demonstrations using proprioception sensor data, an LLM is used to generate behavior descriptions. These descriptions are subsequently used for policy and classifier learning.

Setup. We compare BLADE against the VLM-based baseline Robot-VILA. We omit SayCan and T2M-Shooting since they require additional training data. We first test the original

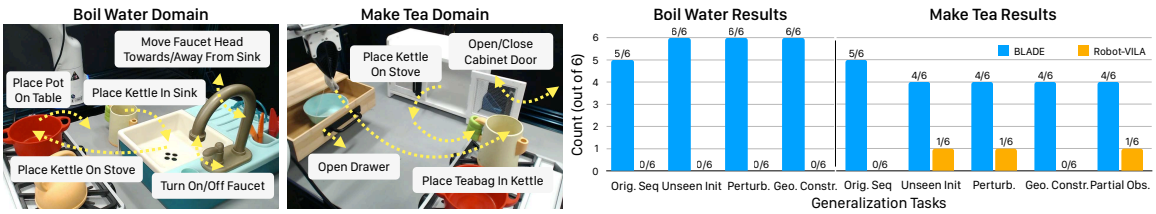


Figure 11-15: **Domains and results on real-world evaluations.** Domain **Make Tea** features a toy kitchen designed to simulate boiling water on a stove. The robot must assess the available space on the stove for the kettle. It also needs to manage the dependencies between actions, such as the faucet must be turned away before the kettle can be placed into the sink to avoid collisions. Domain **Boil Water** involves a tabletop task aimed at preparing tea, incorporating a cabinet, a drawer, and a stove. The robot must locate the kettle, potentially hidden within the cabinet, and a teabag in the drawer. Additionally, it must consider geometric constraints by removing obstacles that block the cabinet doors. In both environments, our model significantly outperforms the VLM-based planner Robot-VILA.

action sequences seen in the demonstrations for each domain. We then test on tasks that require novel compositions of behaviors for four types of generalizations, i.e., unseen initial condition, state perturbation, geometric constraints, and partial observability. For each generalization type, we run six experiments and report the number of experiments that have been successfully completed. See Appendix L.4 for details.

Results. In Fig. 11-15, we show that our model is able to successfully complete at least 4/6 tasks for all generalization types in the two different domains. In comparison, Robot-VILA struggles to generate correct plans to complete the tasks. In Fig. 11-16, we visualize the generated plans and execution traces of both models. In example (a), we show that BLADE can find the kettle initially hidden in the cabinet and then complete the rest of the task. In comparison, Robot-VILA directly predicts placing the teabag in the kettle when the kettle is not visible, resulting in a failure.

11.5 Related Work

Integrated model learning and planning. Integrated model learning and planning is a promising strategy for building robots that can generalize to novel situations and novel goals. Specifically, Chiappa *et al.* [2017]; Zhang *et al.* [2021]; Schrittwieser *et al.* [2020] learn dynamics from raw pixels; Jetchev *et al.* [2013]; Pasula *et al.* [2007]; Konidaris *et al.* [2018]; Chitnis *et al.* [2022]; Bonet and Geffner [2020]; Asai and Muise [2020]; Silver *et al.* [2021] assume access to the underlying factored states of objects, such as object colors and other physical properties. Our work bridges the gap between two groups: we do not

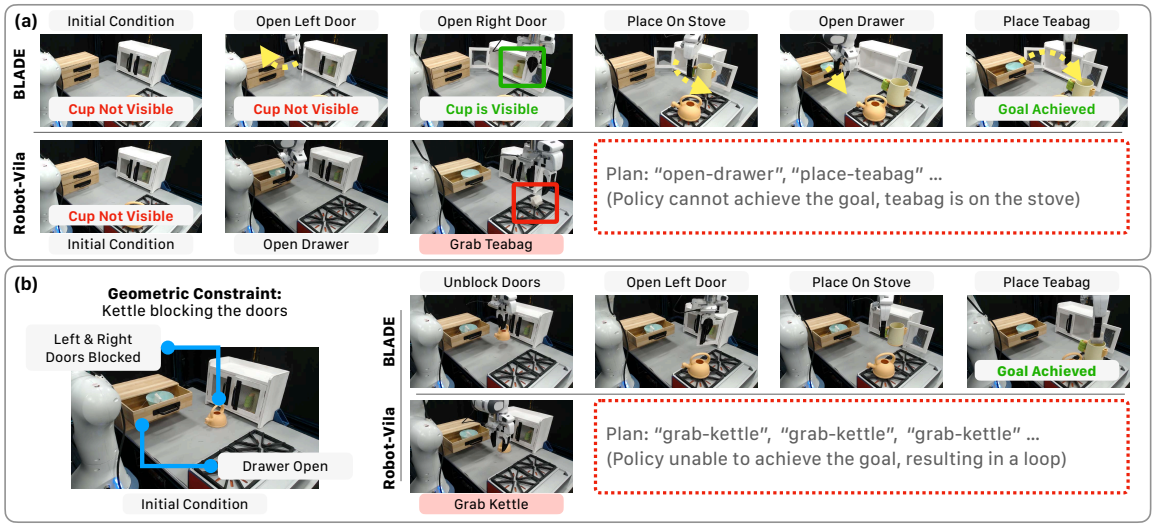


Figure 11-16: **Real world planning and execution.** We show the execution traces from BLADE and Robot-VILA for two generalization tasks: (a) partial observability and (b) geometric constraints.

assume pre-factored state representations given as the input, but learn to ground different factors of object states. More importantly, instead of relying on off-the-shelf models for predicting pixels or for learning first-order rules in symbolic domains, our work considers how human-programmed locality and sparsity structures can improve the model learning and planning efficiency.

Object-factored transition models. Our model learns an object-factored transition model, which generally falls into the category of learning object-oriented MDPs [OO-MDPs; Guestrin *et al.*, 2003; Diuk *et al.*, 2008]. OO-MDPs have been applied to neural network-based representations for visual domains [Walsh, 2010; Kansky *et al.*, 2017; Zhu *et al.*, 2018; Xia *et al.*, 2019; Veerapaneni *et al.*, 2020] and textual game domains [Liu *et al.*, 2021a]. Our model leverages object-centric representations to encode permutation-invariant structures. Furthermore, we exploit fine-grained local and sparse structures in model learning.

Factored model learning. The objective of model learning and planning is to obtain a goal-conditioned policy [Kaelbling, 1993b,a]. While many others have studied model-free [Dayan and Hinton, 1992; Schaul *et al.*, 2015] or hybrid model-free and model-based approaches [Pong *et al.*, 2018; Nasiriany *et al.*, 2019], in this chapter, we focus on learning *structured* models from data and leveraging domain-independent planners for planning in latent representations. Such structured models support novel goal specifications via a first-order logic language, and domain-independent heuristics to accelerate search. Another alternative approach towards learning and planning is to learning to predict subgoals that

needs to be achieved before other goals [Xu *et al.*, 2019a]. However, their work assumes that all preconditions and effects can be represented in a predefined symbolic language, and there are controllers for achieving individual subgoals. By contrast, PDSketch supports generic neural-network-based representations for predicates and action effects.

Neuro-symbolic programs. Our framework PDSketch combines model definition in a structured language (e.g., first-order logic) and neural network learning. It is closely related to the idea of neuro-symbolic programming and differentiable programming for relational reasoning [Chapter 2; Chapter 6; Manhaeve *et al.*, 2018; Riegel *et al.*, 2020; Huang *et al.*, 2021] and policy learning [Sun *et al.*, 2019a]. Our language PDSketch borrows ideas from earlier work on "soft" execution of logical formulas but works on planning domains.

Composing skills for long-horizon manipulation. A large body of model-based planning methods use manually-defined transition models [Srivastava *et al.*, 2014; Dantam *et al.*, 2016; Toussaint, 2015; Garrett *et al.*, 2020a; Curtis *et al.*, 2022; Driess *et al.*, 2020; Yang *et al.*, 2023a] or models learned from data [Finn and Levine, 2017; Nair and Finn, 2020; Shi *et al.*, 2023; Simeonov *et al.*, 2020; Lin *et al.*, 2022; Du *et al.*, 2024] to generate long-horizon plans. However, learning dynamics models with accurate long-term predictions and strong generalization remains challenging. A related direction is to introduce hierarchical structures into the policy models [Luo *et al.*, 2024; Shi *et al.*, 2024; Pirk *et al.*, 2020; Wang *et al.*, 2023a; Lynch and Sermanet, 2021; Mees *et al.*, 2023; Mandlekar *et al.*, 2023], where different methods can segment continuous demonstrations into short-horizon skills [Zhang *et al.*, 2024b; Zhu *et al.*, 2022; Lynch and Sermanet, 2021]. Facing the challenges in modeling action dependencies, these methods are limited to following sequentially specified subgoals. Some work addresses this issue by learning the dependencies between actions from data, but they require large-scale supervised datasets [Zhu *et al.*, 2020; Huang *et al.*, 2019a,b, 2024b]. Our approach is related to methods that learn symbolic action representations [Konidaris *et al.*, 2018; Silver *et al.*, 2023; Ahmetoglu *et al.*, 2025; Shah *et al.*, 2024; Han *et al.*, 2024]; the difference is that BLADE uses a LLM to generates causal models of the environment and learns their groundings on sensory inputs.

Using LLMs for planning. Many researchers have explored using LLMs for planning. Methods for direct generation of action sequences [Huang *et al.*, 2022b; Silver *et al.*, 2022] can struggle to produce accurate plans [Valmeekam *et al.*, 2023; Kambhampati *et al.*, 2024]. Researchers have also leveraged LLMs as translators from natural language instructions to symbolic goals [Chen *et al.*, 2024; Liu *et al.*, 2023a; Xie *et al.*, 2023; Mavrogiannis *et al.*, 2024], as generalized solvers [Silver *et al.*, 2024], as memory modules [Zhu *et al.*,

2023a], and as world models [Nottingham *et al.*, 2023; Hao *et al.*, 2023]. To improve the planning accuracy, prior work has explored techniques including using programs [Liang *et al.*, 2023; Singh *et al.*, 2023], learning affordance functions [Ahn *et al.*, 2022; Lin *et al.*, 2023], replanning [Skreta *et al.*, 2024], finetuning [Driess *et al.*, 2023; Wu *et al.*, 2023; Xiang *et al.*, 2024], embedding reasoning in a behavior tree [Wang *et al.*, 2024a], and VLM-based decision-making [Hu *et al.*, 2023; Wake *et al.*, 2023]. BLADE shares a similar spirit as methods using LLMs to generate planning-compatible action representations [Wong *et al.*, 2024; Guan *et al.*, 2023; Smirnov *et al.*, 2024]. However, they make assumptions on the availability of state abstractions, while BLADE grounds LLM-generated action definitions without additional labels. Also complementary to methods that leverage these representations for skill learning [Li *et al.*, 2024; Dalal *et al.*, 2024], our approach uses them for composing skills in novel ways.

11.6 Discussion

This chapter demonstrates two complementary approaches for advancing neuro-symbolic long-horizon planning. PDSketch shows that explicitly encoding structural priors (locality and sparsity) into planning models yields strong data efficiency, compositional generalization, and powerful domain-independent heuristics. BLADE extends these ideas by showing how such domain specifications can be acquired automatically, through the interplay of demonstrations, language, and large language models.

Together, they highlight a broader principle: effective robot planning requires both well-structured representations and scalable methods for acquiring them. By integrating model-based planning with learned abstractions, these approaches bring us closer to building general-purpose robots that can continually learn, generalize to novel environments, and robustly execute long-horizon tasks in the real world.

So far, one limitation of the current system (BLADE) is that the automatic segmentation of demonstrations is based on gripper states; more advanced contact detection techniques might be required for certain tasks such as caging grasps. We also assume the knowledge of a given set of predicate names in natural language and focus on learning dependencies between actions using the given predicates. Automatically inventing task-specific predicates from demonstrations and language annotations, possibly with the integration of vision-language models (VLMs) is an important future direction. In our experiments, we also found that noisy state classification led to some planning failures. Therefore, developing planners that are more robust to noises in state estimation is necessary. Finally, achieving novel compositions of behaviors also requires policies with strong generalization to novel environmental states, which remains a challenge for data-efficient skill learning.

Chapter 12

Hybrid Imperative-Declarative Action Representations

The final technical chapter of this dissertation presents a novel representational and reasoning framework that integrates many of the key insights developed throughout the dissertation: neuro-symbolic logic reasoning, constraint satisfaction solving, concept learning, skill learning, and planning, with a particular focus on robotic planning problems. Specifically, we extend the planning-centric framework from Chapter 11 into a hybrid imperative–declarative decision-making framework. Rather than treating declarative planning and imperative policy execution as separate approaches, we unify them into a single representation. This design is inspired by the theoretical findings of Chapter 4 on reasoning about discrete actions, as well as the conceptual framework introduced in Chapter 5 for reasoning about continuous values. Taken together, this representation and reasoning framework opens up new directions for learning neuro-symbolic action representations and planning with them efficiently.

Technically, we present a robot-behavior description language CDL that can express both direct imperative strategies and planning-based strategies, and combine them seamlessly within the same program. Accompanying this language is a general-purpose planner CROW, which interprets the behavior description and searches as necessary to find a sound plan. We demonstrate (1) via example programs, that CDL can be used to specify, very intuitively, different known strategies for navigation among movable obstacle (NAMO) problems, (2) via empirical results, that CROW can take advantage of the priors expressed in CDL to very quickly solve problem instances with known simplifying structure but still generalize to hard instances, and (3) via theory, that *width*, a powerful characterization of the worst-case complexity of planning problems, corresponds to a natural property of CDL descriptions and that CROW operates in time on the same order as the width-based worst-case complexity.

The material of this chapter has been previously published in Mao *et al.* [2024].

(a) The "direct connect" approach.

```
behavior achieve_at_1(g: Pose):
  goal: at(g)
  body:

    bind t: Trajectory where:
      connect(s, g, t)


    forall o: Object:
      achieve not blocking(o, t)
    do exec_traj(t)
```

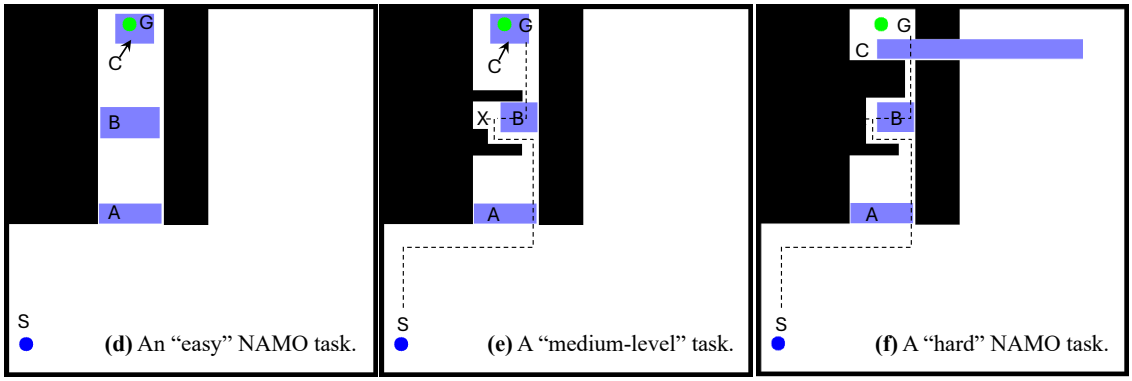
(b) Searching for waypoints.

```
behavior achieve_at_2(g: Pose):
  goal: at(g)
  body:
    bind x: Pose # intermediate pos
    bind t: Trajectory where:
      connect(x, g, t)

    achieve at(x)
    forall o: Object:
      achieve not blocking(o, t)
    do exec_traj(t)
```

(c) Searching for waypoints and orders.

```
behavior achieve_at_3(g: Pose):
  goal: at(g)
  body:
    bind x: Pose
    bind t: Trajectory where:
      connect(x, g, t)
    promotable:
      achieve at(x)
      forall o: Object:
        achieve not blocking(o, t)
    do exec_traj(t)
```



(d) An "easy" NAMO task.

(e) A "medium-level" task.

(f) A "hard" NAMO task.

Sol: the robot can drag object A then object B then object C into the empty space to clear a path to reach goal point G.

Sol: the robot moves to location X (after clearing A), then drags obstacle B downward to clear the path from X to G, then moves to G.

Sol: the robot needs to work at beginning of the trajectory, to enable traversal of the end of the trajectory: it first pulls C out to the right, then drags A into free space to reach X, then moves B down and can reach G.

Figure 12-1: **NAMO problems and solutions.** (a)–(c):CDL specifications for three NAMO solutions; (d)–(f): example problems of different degrees of difficulty. See Appendix M.4.

12.1 Introduction

Hybrid discrete-continuous decision problems are critical in robotics. The basic pick-and-place object manipulation is a classic example, where the robot needs to produce a sequence of primitive actions that are parameterized by both discrete variables (objects to manipulate), and continuous parameters (grasping poses, trajectories, etc.). This problem formulation extends to rich problem classes involving non-prehensile manipulation, articulated object manipulation, etc.

In this chapter, we consider such hybrid decision problems with discrete-time transitions. As algorithm developers, we must provide an algorithm for the robot to take the current world state (or, in general, a belief state) and produce the next action. Classically there are two main strategies. In *imperative* strategies we construct a direct mapping, in the form of a computer program or neural network weights with no search. Optionally, the mapping can be hierarchical, where a sequence of functions needs to be called as nested subroutines to generate the first action. These approaches use a roughly constant amount

of computation (independent of problem size, horizon, etc.) to determine the next action. Typical neural network (hierarchical) policies, behavior trees, PID controllers, etc., fall into this category. In the second *declarative* strategies, we compute the next action indirectly by first specifying a space of possible solutions, such as all possible sequences of actions and we "search" in this space to find a sequence that satisfies state-dynamics constraints (in terms of a transition model) and that reaches a design goal while minimizing a cost objective. Classical AI planners, task and motion planners, and model-predictive controllers all fall into this category.

There are important trade-offs between these approaches, such as the compactness of the representation and the inference runtime, but they are *not* problem-independent. For some problem classes, there are compact imperative programs that can be directly written by humans or easily learned via imitation or reinforcement learning. At the other extreme, for some problem classes, there is no compact imperative representation (e.g., for PSPACE-Hard problems, see also Chapter 4 and Vega-Brown and Roy [2020]). For such problems, the most effective approach may be to specify or learn a declarative transition model and use it at performance time to search. Some other approaches such as hierarchical task networks [HTNs; Sacerdoti, 1975] and hierarchical planning in the now [HPN; Kaelbling and Lozano-Pérez, 2011] occupy a middle ground between completely imperative and completely declarative methods. They leverage some heuristics, such as imperative structures and execution constraints to reduce runtime complexity at the cost of some loss in generality.

However, one major drawback of almost all of the existing methods for behavior generation is that they take a *uniform* approach: they are completely imperative or completely declarative or completely follow the restrictions of STRIPS. But, as we try to address bigger and more general problems (such as designing a general-purpose household assistant robot that can operate over a horizon of days), we will want to "mix and match" these approaches.

The key insight that allows us to unify and combine these approaches is that the differences among them can be described by different types of *branches and commitments*, such as commitments to the order for achieving subgoals, the choice of plans to achieve a particular subgoal, and the possible interleavings of subgoal achievements. A generic formulation for a *behavior* is to think about it as a nondeterministic program that can generate a sequence of control commands to achieve certain goals. Such programs may involve recursive function calls to other behaviors to generate part of the control commands for some "subgoals." In the completely imperative case, we have an action generator that already has all correctness conditions built into it, so that we can immediately commit to a single choice at every turn. In the completely declarative case, we cannot commit to executing any action until we have generated an entire sequence and verified its soundness. In section 12.4 we detail structural assumptions in the middle ground made by several existing behavior specification methods.

NAMO examples Consider the motivating example of a NAMO [navigation among movable obstacles; Stilman and Kuffner, 2005] domain illustrated in Fig. 12-1, where the task is to move to a target location potentially moving obstacles in order to reach it. In Fig. 12-1, we describe a series of solution strategies, using our proposed behavior description language CDL. The core of a CDL program is a set of *behavior rules* that specify methods for achieving some goal. These rules may provide a sequential deterministic program for achieving the goal or may describe a decomposition into one or more abstract or concrete steps without specifying exactly how to do them, in what order, or with what variable bindings.

Fig. 12-1a describes a hierarchical policy that is almost fully imperative: find a path that connects the current robot position s to the goal g (`bind` statement), move each obstacle out of the way (via another behavior rule, specified in the `achieve` statements), and then execute a controller to traverse the path (`do` statement). This strategy will work for simple problem instances, such Fig. 12-1d, where the obstacles can be moved individually and there is sufficient additional storage space. Fig. 12-1b describes a slightly more complex strategy, inspired by Stilman and Kuffner [Stilman and Kuffner, 2005]: pick a sequence of waypoints, then traverse to the waypoints, clearing each path segment before it is traversed. Example Fig. 12-1e, is solvable with this method but not with method in Fig. 12-1a, because it is not possible to clear the entire path at once. Fig. 12-1c describes a more general method: like the previous method, it states that we will have to reach a sequence of waypoints, and that the path segments must be cleared, but (via the `promotable` section) it gives the planner the freedom to do these steps in any order, and even to interleave the sub-plans for achieving the subgoals. It solves all problem instances, including Fig. 12-1f, which cannot be solved by the previous two algorithms. The important observation here is that it is easy, in CDL, to specify a whole range of algorithms in a very clear and concise way.

In summary, our contributions in this chapter are:

1. The CROW definition language (CDL), a formalism for specifying robot behavior in hybrid discrete-continuous domains that encompasses imperative and declarative specifications, as well as a highly flexible range of combinations. Furthermore, it enables easy specification (by humans) of domain knowledge that, for example, can reduce the difficulty of a worst-case PSPACE-hard planning problem to a lower-order polynomial by constraining how subtask solutions can be committed independently and serialized.
2. The CROW interpreter for CDL, a behavior-generation algorithm that takes a behavior specification and implements a policy that maps domain states into actions, with an internal computation structure ranging from direct policy-following to complex planning with backtracking and constraint-based variable binding, based on the specification.

12.2 The Crow Definition Language (CDL)

CDL is a flexible language for describing goal-directed behavior for discrete-time hybrid discrete-continuous space decision-making. At performance time, a CDL interpreter takes a CDL behavior specification, the current state, and the goal as its input. It translates the raw sensory input into a structured representation, then recursively composes behavior rules to find a sequence of control commands that can achieve the goal. Finally, it executes the generated control commands.

A hybrid decision-making problem is a tuple of $\langle \mathcal{S}, C, \mathcal{T}, s_0, g \rangle$, where $\mathcal{S}$ is the set of states, C is a set of primitive controllers, $\mathcal{T} : \mathcal{S} \times C \to \mathcal{S}$ is a (deterministic) transition function, s_0 is the initial state and $g : \mathcal{S} \to \{0, 1\}$ is a goal classifier. It is hybrid because elements in both $\mathcal{S}$ and C have discrete and continuous values in different "dimensions" (e.g., a set of grasping controllers might be parameterized by $\mathcal{U} \times \mathbb{R}^6$ where $\mathcal{U}$ is the universe of entities in the scene and $\mathbb{R}^6$ denotes grasping poses). In this chapter, we only consider tasks where the states are fully observable, transitions are deterministic, and actions are of uniform costs. The task is to find a sequence of primitive controller actions $\bar{a} = \{a_1, \cdots a_T\}$, which are functions that can be executed in the (physical) world. We want the sequence $\bar{a}$ to be *sound* in the sense that $g(s_T)$ is true (goal is reached) and $s_t = \mathcal{T}(s_{t-1}, a_t)$ (state transitions are satisfied) in the world.

Fundamentally, we can view CDL as specifying a *generative model* for sound primitive plans, together with language constructs that can dramatically improve the computational efficiency of finding such plans. The space of plans is generated via non-deterministic choices of methods for achieving subgoals (described in section 12.2.2), of ordering and interleaving of subgoals (described in section 12.2.3), and of values for free variables (described in section 12.2.4). This is generally an infinite space of possible plans, which requires intelligent search. In section 12.2.2 we show that, by characterizing action effects and asserting conditions that must hold at various points in the plan, we can substantially prune the search space. In section 12.2.5, we show that, by committing early to some non-deterministic choices, we can get exponential reductions in the space. We include more details and examples in Appendix M.4 and Appendix M.5.

Interface to perception and control We begin by defining the input and output interfaces between CDL and the robot for which it is specifying behavior.

For planning purposes, *states* ($\mathcal{S}$) are represented in a relational manner, in terms of entities and relational features associated with them. Formally, a state is represented as a tuple $s = \langle U, F \rangle$, where U is the universe of entities, assumed to be a fixed finite set. F is a set of relational features that can take discrete or continuous values. All entities and values

are typed in CDL. Each feature $f \in F$ can be viewed as a table where each entry is associated with a tuple of entities $(o_1, \cdots, o_k)$. Each entry has the value of the feature in the state. For example, in NAMO, we define the feature `position(x)` as a unary feature of type `Pose`.

```
typedef Object
typedef Pose: vector[float32, 2]
feature agent_position() -> Pose
feature position(x: Object) -> Pose
feature holding(x: Object) -> bool
```

We assume that an external "perception" system can segment raw sensory input into individual objects and that, for each feature, there is a program defined in an external language (e.g., Python) that can compute the feature value for any tuple of objects of the appropriate arity.

Actual behavior in the world is generated through *controllers*, which are primitive functions that take entities and values as inputs and execute a low-level sensorimotor loop to effect the world, returning control when they have completed. Examples include tracking a motion trajectory or grasping an object.

```
controller exec_traj(t: Trajectory)
controller attach(x: Object)
controller detach(x: Object)
```

Behavior rules The primary part of a CDL program is a set of *behavior rules* $\mathcal{B}$. Primitive plans are generated through a refinement process, which starts with the top-level goal and repeatedly applies behavior rules to a partial plan until it is refined into a completely primitive plan. In CDL, each behavior rule specifies a strategy for achieving the condition that some state feature, applied to some objects, has a desired value (examples might be `holding(o)` or `position(o) == p`). Importantly, they are "lifted" in the sense that the goals they achieve are parameterized by variables. Thus, a behavior rule for `holding(o)` can be used to achieve the holding feature of any object.

Behavior rules play multiple roles in CDL, as we will show in subsequent sections. In their simplest form, each behavior rule is a tuple of $\langle name, args, g, B \rangle$, where *name* is the name, *args* is a set of parameters, g is a goal statement (a Boolean expression over state features) that will be achieved after successfully executing B, and B is the body, which is a non-deterministic program that, through refinement, can generate primitive plans.

12.2.1 Imperative Behavior Scripting in CDL

The simplest style of behavior specification that CDL supports is deterministic hierarchical policies. In this case, the body $body(b)$ for each behavior rule $b \in \mathcal{B}$ is a deterministic program with subroutines, consisting of a sequence with three types of statements. First,

```
let v = f(v, ...)
```

declares a local variable `v` and assigns a value to it, which is the return value of `f`. f is a function defined in an external language and can take values of the program parameters, and values in previous `let` statements as input. Next,

```
achieve f(...)
```

recursively calls another behavior rule whose goal is the feature `f` applied to variables available in the current rule. Finally,

```
do c(...)
```

inserts into the plan being constructed a call to primitive controller `c` parameterized by some variables in the current rule.

The formal semantics of these statements will be defined later when we describe the plan refinement processes. Here, for illustration purposes, we consider these constructs in the following simple program for clearing a movable obstacle `o` out of the way of a specified trajectory `t`.

```
behavior move_away(o: Object, t: Trajectory)
  goal: not blocking(o, t)
  body:
    # Move the agent to a position where it is close to o
    achieve close_to(o)
    do attach(o)
    # Find a free location for the obstacle o away from t
    let o_pos = free_location(o, t)
    achieve position(o) == o_pos
    do detach(o)
```

This program is fully imperative. Given the target object `o` to move and the trajectory `t` to clear, it begins by calling another rule to achieve the subgoal `close_to(o)`. This will construct the first part of a primitive plan. Then it inserts a call to a primitive controller to `attach(o)`. Next, it has to find a place to put the object (for instance in some designated storage place) using the `let` statement. It calls another rule to generate steps to put the object into position, and adds the final primitive controller to detach the object.

This rule is simple, but can only solve a very limited set of NAMO problems, in which there is plenty of easily accessible storage space so that we do not need to coordinate the placement of objects into it, and where moving each object out of the trajectory does not require moving any other objects.

In the case where $\mathcal{B}$ consists of behavior rules of this type, and where there is at most one $b \in \mathcal{B}$ with the same feature f as its goal, $\mathcal{B}$ deterministically specifies a single primitive plan for an initial goal g. This is effectively the same as a simple type of behavior tree [Colledanchise and Ögren, 2018]. Such rules can be executed very efficiently, but they put the entire burden of behavior specification on the programmer. The interpreter *cannot* check the resulting primitive plan for soundness, and relies entirely on the programmer to construct a correct plan.

12.2.2 Nondeterminsitic Behaviors and Soundness Checking

Fully deterministic behavior rules are hard to write in general, especially for large and complex problems involving many different goals and subgoals spanning over a long horizon. Therefore we transfer some of the burden to the CDL interpreter, by letting the programmer specify a space of possible plans (via non-deterministic behavior rules) and letting the interpreter implement a search to find one that satisfies the goal, as well as possibly other intermediate soundness checks.

There are several types of nondeterminism available in CDL; in this section, we begin with the simplest, in which we allow $\mathcal{B}$ to contain multiple behavior rules with the same goal feature g. For example, there may be other methods for achieving `not blocking(o, t)`, such as sweeping it away with a broom, that require very different action sequences and might have different additional effects (for example, that the robot is holding a broom, or the object is toppled over), which might affect the selection of later steps in the plan.

The programmer now has the freedom to describe many possible action sequences, but we must also characterize, implicitly, which ones will be successful in achieving a desired result. To enable this, we introduce two additional language features: `assert` statements and the `eff` (effect) section. The statement

```
assert e(...)
```

requires that an expression should evaluate to true at a given step in the execution of a plan. The expression `e` can be made up of Boolean combinations of features or be the application of an externally defined function to variable values or features of objects in the rule. The effect section of a behavior rule

```
behavior <name>(<args>):
  eff:
```

```
3       f[...] = ...
4       forall x: Object: g[x, ...]= ...
```

is a short program that describes changes to the state that result from applying the behavior rule. It can contain two types of primitive statements: feature assignments and `for` loops over all objects of a certain type. The value to be assigned can be determined by any external function applied to variable values or features of objects in the rule. Consider the example of a behavior rule that moves objects. It may specify its effects on both the position of the agent and the object. It may begin by asserting a precondition that the robot be holding the object. Including such preconditions is not strictly necessary for correctness, but they have a huge effect on efficiency, as we will see.

Formally, we model the semantics of a set of CDL rules $\mathcal{B}$ in terms of a set of candidate *plan traces* that can be produced by using $\mathcal{B}$ to *refine* an initial goal g. A plan trace is a sequence of three types of statements: `do` statements, `assert` statements, and `eff` statements. The refinement process for goal g begins with an initial *unrefined plan trace*, which consists of two statements, $\{\texttt{achieve}(g); \texttt{assert}(g)\}_{seq}$. This initial trace will be non-deterministically refined, by using rules in $\mathcal{B}$ to replace `achieve` statements with subsequences of other statements, such as refining an `achieve close_to(o)` with a sequence of primitive movement actions, until we have a fully refined plan trace with only `do`, `assert` and `eff` statements. We define the non-deterministic plan refinement operator `RefineS` as: where $P_{<i}$ denotes the prefix of P preceding index i while $P_{>i}$ denotes the

Algorithm 9 The algorithmic definition of the simple refinement operator $\mathcal{R}_0$ for sequential body programs. $P_{<i}$ denotes the prefix of the program P preceding index i while $P_{>i}$ returns the suffix. $\oplus$ is the sequence concatenation operation.

function RefineS(P)
  if there are no `achieve` statements in P **then return** P
  else
    Let g, i be the goal and index of the last `achieve` statement in P
    for $b \in \{\mathcal{B} \mid goal(b) = g\}$ **do**
      yield RefineS($P_{<i}$) $\oplus$ RefineS($body(b)$) $\oplus$ {`eff` $eff(b)$} $\oplus$ $P_{>i}$

suffix, and $\oplus$ is the sequence concatenation operation. (Note that, in general, we compute a substitution of variable names necessary to match g with $goal(b)$, and apply it to $body(b)$ and $eff(b)$ before adding them to the refined plan trace.) We borrow the "yield" syntax from Python to define a generator function. Each time the "yield" statement is hit, it produces a plan trace that an outer for-loop such as "**for** $P \in$ RefineS(P_0) **do** ..." can consume.

The refinement operation has the potential to produce a large number of plan traces when some `achieve` statements have multiple possible refinements. Through assertions and effect declarations, $\mathcal{B}$ provides internal *soundness* constraints on primitive plans that

can be generated. First, given a plan trace $\bar{a} = \{a_1, a_2, \cdots, a_k\}_{seq}$, and an initial state s_0, at each step $t = 1, 2, \cdots, k$, we define the *projected state* $\hat{s}_t$ as: $\hat{s}_0 = s_0$,

$$\hat{s}_t = \begin{cases} update(\hat{s}_{t-1}, \phi) & \text{if } a_t = \texttt{eff}\ \phi \text{ is the effect of behavior } \phi \\ \hat{s}_{t-1} & \text{otherwise} \end{cases}$$

where $update(\hat{s}, \phi)$ is the result of assigning, in $\hat{s}$, all the feature-value pairs specified in ϕ. A plan trace $\{a_1, a_2, \cdots, a_k\}_{seq}$ is *internally sound* if and only if all asserted conditions evaluate to true at their projected states; that is, for all $t <= k$, if $a_t = \texttt{assert}\ \psi$ then $eval(\psi, \hat{s}_t)$, where $eval(\psi, s)$ is true if the Boolean statement ψ is satisfied in state s. Ultimately, we would like to find plans that are *externally sound with respect to a goal g*; that is, guaranteed to drive the world into a state satisfying g.

Proposition 12.2.1. *Internal soundness implies external soundness w.r.t. g when (1) each plan trace ends with a final assertion that g holds; and (2) the effect statements of all behavior rules are consistent with the true world transitions.*

In such cases, executing the sequence of primitive controllers in an internally sound plan trace will achieve the intended goal in the world. Although it may sound difficult to state the effects of all primitive actions, one simple implementation strategy, assuming access to a resettable simulator representing the underlying transition model, is to use the simulator to compute the next state.

12.2.3 Subgoal Ordering and Promotion

In this section, we enrich the flexibility of the plan generator with two more types of non-determinism: subgoal ordering and interleaving. Consider a situation in which a robot has to place an object in a cupboard: clearly, before it places the object, the cupboard door should be open and the robot should be holding the object. But in which order should it achieve those preconditions? It might depend on how many hands the robot has, the distance between where the object has to be picked up and the cupboard, what kind of handle the cupboard has, etc. It is impossible to pick a good static ordering of these operations and much too difficult for the programmer to write a conditional expression to decide which to do first. So, we can allow the CDL generator to consider different orderings of the subgoals and find one that yields a sound plan. Furthermore, imagine that opening the cupboard requires retrieving a step-stool first. Now, it might turn out that we need to actually interleave the primitive steps for opening the cupboard with those for picking up the object.

To extend CDL to handle these cases, we add three new language features, extend the structure of partially refined plan traces, and provide a more sophisticated refinement algorithm.

Unordered statement groups We allow a behavior rule body to have multiple `unordered` sections, containing `achieve` and `do` statements. This is simply equivalent to having multiple fully ordered rules for the same goal, one for each possible choice of ordering of each unordered group. A common case in which we may wish to have unordered groups is when we wish to achieve some subgoal with respect to all objects of some type. We add a section type `foreach` that provides unordered iteration over those objects. For example,

```
1  unordered:
2    statement_1(); statement_2(); ...
3  foreach o: T:
4    statement_3(o); statement_4(o); ...
```

where ... consists of the same types of statements as in a rule body. Here, the order for statements 1 and 2 are unordered. For each object o of type `T`, the order for executing statements 3 and 4 on them, is also undetermined.

Promotable statement groups Unordered `achieve` statements may be refined into plans in any order, but the corresponding primitive controller actions may not be interleaved. For most planning problems, this is sufficient flexibility, but sometimes more is necessary. To accommodate such problems, we extend the syntax of the body of a behavior rule to allow one set of statements, called a `promotable` section, (typically only `achieve`) to be unordered *and* to have their achievements interleaved.* The promotable section may be preceded and followed by additional ordered or unordered lists of statements of any type.

For example, consider the behavior rule `achieve_at_3` in figure 12-1(c). The `promotable` section says that the robot has to get to waypoint `x` and ensure that the rest of the trajectory to get from `x` to the goal location `g` is clear. These things may need to happen in any order (in our example, it has to clear part of the trajectory before moving to `x`; more generally, it might need to further interleave steps for moving the obstacles out of the way).

Plan traces and refinement To generate all plan traces specified by this more expressive class of behavior rules, we adopt a more general view of partially refined plan traces and refinements. We describe this in Appendix M.1.

First, in general, partially refined plan traces will not be totally ordered. Instead, they will be partially ordered, and represented using nested ordered and unordered sequences. For

*We support statements that are ordered and interleavable, but this is less useful.

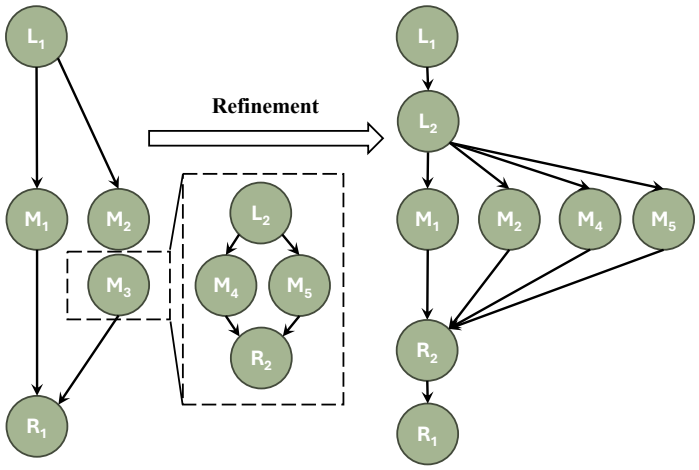


Figure 12-2: **Illustration of the refinement operator.** Each node corresponds to a statement (or a sequential program). We first nondeterministically select a node in the "middle" section that is the last statement of one of the sequential chains, and then nondeterministically select a behavior to refine it. Finally, we construct a new plan by "inserting" the refined sub-plan into the graph following the rules.

example, the ordering $\{\{p_1, p_2\}_{seq}, \{p_3, p_4\}_{seq}\}_{unordered}$ states that p_1 will be executed before p_2 and that p_3 will be executed before p_4. However, the relative ordering between, for example, p_1 and p_3 is not specified.

Next, we observe that each rule body can be expressed as $\langle L, M, R\rangle$ with three sections: left (everything before the promotable section), middle (the promotable section), and right (everything after the promotable section). Any of these sections may be empty. In general, the L and R sections may be partially ordered and the M section is unordered. We will adopt this same structure for partially refined plan traces, with the initial plan trace having the form $\langle \emptyset, \emptyset, \{\texttt{achieve}(g), \texttt{assert}(g)\}_{seq}\rangle$.

The generalized plan refinement operator $\mathcal{R}^* : \mathcal{P}^3 \to \mathcal{P}_0$ takes as input a partially refined plan trace and non-deterministically refines it into a plan trace $\bar{a}$. $\mathcal{R}^*(P)$ recursively replaces one of the `achieve` statements in P with the body of a behavior b whose goal matches the achieve statement. To handle possible orders of achieving subgoals, it non-deterministically selects a statement β that could be the last statement in L, M, or R (see Appendix M.1). If β is not an `achieve` statement, it is directly inserted into the plan. Otherwise, we non-deterministically select a behavior b whose goal matches β to handle multiple behavior rules for goal β.

To handle interleaved execution, we consider two cases: when β is from M and when β is not. Let $\langle L', M', R'\rangle$ be the three sections of $body(b)$. When β is not from M, we simply refine $\langle L', M', R'\rangle$ recursively and concatenate the resulting plans. When β is from M, statements in L' will be appended to L, those in R' will be prepended to R, and M will

be unioned with the rest of M (i.e., $M \setminus \{\beta\}$, where the $\setminus$ operator (set difference) returns a partially ordered plan just as M but with β removed). Figure 12-2 illustrates the operation on the partially ordered graph. The key insight here is that if q is a subgoal in a partially ordered plan β, we do not assume knowledge of how M' and the rest of the subgoals in $M \setminus \{\beta\}$ can be ordered.

This new rule for promotable sections is inspired by concepts from hierarchical planning and *subgoal serialization*. In conventional hierarchical task and goal networks [Nau *et al.*, 2003; Alford *et al.*, 2016], the "bodies" from two different behaviors are always unordered. This is equivalent to marking all statements as promotable. In contrast, planning can be exponentially more efficient when subgoals can be serialized (Chapter 4), which is the behavior of the "left" and the "right" sections. CDL unifies these two types of behavior rules allowing the programmer fine-grained control to obtain just enough generality while retaining as much efficiency as possible.

12.2.4 Variable Binding

In the examples we have seen so far, we have had variables that are deterministically bound using `let` statements. In most robotics domains, however, we need to choose values from large or infinite sets (of configurations, grasps, poses, trajectories, etc.) and a crucial part of the planning process is selecting these values to construct a sound plan.

Some task and motion planning systems view the problem of variable binding as one of constraint satsifaction [Garrett *et al.*, 2018]. We adopt this view, allowing a behavior rule to indicate that the value of a variable can be chosen non-deterministically, possibly subject to a constraint that relates its value to the values of other variables. We introduce a new statement type:

```
1  bind v: T = f(v, ...)
```

which requires that `v` have a value v of type `T` that satisfies a condition `f`, expressed as a function (defined in an external language) of the value v, values of the program parameters, and values in previous `bind` statements. We have an example in the behavior rule `achieve_at_3` in figure 12-1(c): The first type of `bind` statement for `x` only constrains the variable by its type. The second one for trajectories can be reduced to two statements:

```
1  bind t: Trajectory; assert connect(x, g, t)
```

However, for the purposes of solving the embedded constraint-satisfaction problem, it is useful to keep the variable-value constraints separate. It is also important to note that in general, constraint functions should be accompanied by conditional generators that can produce satisfying values of the variable being bound given values for other arguments. In

our example, a motion planner could be used to generate possible trajectories between a given pair of `x` and `g`.

Note that the variable values will generally not be chosen individually when refining bind statements in behavior rules, but that a constraint set will be accumulated and solved jointly later. In particular, when we have `bind` statements, applying the refinement operator on an initial unrefined plan trace will give a set of variable-augmented plan traces. Let the $\langle \bar{a}, V \rangle$ be a tuple of the plan trace $\bar{a}$ with a set of associated variables V. $\langle \bar{a}, V \rangle$ is *sound* if and only if there exists an assignment A (mapping from variables in V to their domains) where $\bar{a}_{V \setminus A}$ is a sound plan. Here, the $\bar{a}_{V \setminus A}$ operation replaces all variables in $\bar{a}$ with their assignments in V.

The ordering of `bind` statements in the final plan suggests a variable ordering for solving the CSP at plan-generation time, although any sound (and ideally complete) CSP-solving method can be used. We include details in Appendix M.2.

12.2.5 Commitment

In some cases, there may be many alternative solutions for a subpart of a problem, any of which will work fine with any solution for the next part. In such cases, it may be very expensive at planning time to maintain all of those alternatives. To allow a programmer to communicate this situation to the interpreter algorithm, we add a `commit` statement, inspired by Prolog's `cut`, to CDL. It does not increase the space of possible plans or change the soundness conditions. If used incorrectly, however, it can endanger completeness by prematurely ruling out a potentially good alternative. To gain intuition, consider a modification to part of the behavior rule `achieve_at_1` in figure 12-1(a) for moving obstacles:

```
forall o: Object:
  achieve not blocking(o, t)
  commit # newly added!
```

Here, we have added a `commit` statement, so that after finding a path for clearing some object $o1$ from the trajectory `t` the interpreter will commit to the first part of the plan trace as well as the continuous variables associated with it (e.g., the new location for the object `o1`). Adding this explicit commit forbids trying different placements for this object, but does not forbid placing the objects in a different order. When there is enough space for reorganizing objects, this behavior can succeed with high probability while significantly improving runtime.

A `commit` statement can appear in the left or right section of a behavior program. Take the left section as an example. When we refine $\langle \{\ldots; \texttt{commit}\}_{seq}, M, R \rangle$, instead of considering

all possible refinements to the rest of the left program $\textsc{refine}(\langle L \setminus \{\beta\}, \emptyset, \emptyset\rangle)$, we instead consider an arbitrary one of them. In practice, this may be implemented by considering only the first plan trace generated from the recursive refinement to possibly prune out a large number of branches — we will see this in action in our Crow interpreter in Section 12.3.

12.3 The Crow Interpreter

In this section, we describe a practical implementation of an interpreter for CDL, called Crow (Compositional Regression and Optimization Wayfinder). It is a non-deterministic algorithm, with a structure similar to the refinement operator, but crucially with pruning based on assertions and commitments.

It keeps track of a program tuple $\langle L, M, R\rangle$, and a set of variables and constraints $\langle V, C\rangle$. It uses the following implementations to handle behavior branching and continuous constraint satisfaction — it is a depth-first search algorithm given a set of behaviors $\mathcal{B}$. Its basic structure is to take the program tuple as input and recursively compute an output tuple consisting of a resulting state s', action sequence $\bar{s}$, variable set V, and constraint set C.

1. If L is not empty, it recursively solves $\langle \emptyset, \emptyset, L\rangle$ first, and then solves $\langle \emptyset, M, R\rangle$.
2. Else, if M is not empty, it recursively solves M and then R. When it is solving M, it will consider merging promotable sections of the new behavior and the current M following the definition of the refinement operators.
3. Else, it solves statements in R one by one and concatenates the plan traces.

It defers the constraint satisfaction solving after checking all purely discrete constraints by tracking variable ordering throughout the expansion.

---

function Plan($state$, g, $\mathcal{B}$) ▷ Top-level entry
  let $P_0 \leftarrow \left\langle \emptyset, \emptyset, \{\texttt{achieve}(g); \texttt{assert}(g)\}_{seq}, \emptyset, \emptyset \right\rangle$
  for $\langle s, \bar{a}, V, C\rangle \in$ Crow($state$, P_0, $\mathcal{B}$) **do**
    if Solve-CSP(V, C) returns an assignment A **then**
      return $\bar{a}_{V \setminus A}$ ▷ We have found a sound plan
function Crow(s, $\langle L, M, R, V, C\rangle$, $\mathcal{B}$)
  if $L \neq \emptyset$ **then**
    for $\langle s', \bar{a}, V', C'\rangle \in$ Crow(s, $\langle \emptyset, \emptyset, L, V, C\rangle$, $\mathcal{B}$) **do**
      for $\langle s'', \bar{a}', V'', C''\rangle \in$ Crow(s', $\langle \emptyset, M, R, V, C\rangle$, $\mathcal{B}$) **do**
        yield $\langle s'', \bar{a} \oplus \bar{a}', V'', C''\rangle$
  else if $M \neq \emptyset$ **then**
    for $\beta \in tail(M)$ **do**

if β is `achieve` **then**
for $b \in \{\langle L', M', R' \rangle \in \mathcal{B} \mid goal(b) = \beta\}$ **do**
let $P' = \langle \{L, L'\}_{seq}, \{M \setminus \{\beta\}, M'\}_{unordered}, \{R', R\}_{seq}, V, C \rangle$
yield from Crow$(s, P', \mathcal{B})$
else yield from Crow$(s, \langle L, M \setminus \{\beta\}, \{\beta, R\}_{seq}, V, C \rangle, \mathcal{B})$
else if $R \neq \emptyset$ **then**
for $\beta \in tail(R)$ **do**
for $\langle s', \bar{a}, V', C' \rangle \in$ Crow$(s, \langle L, M, R \setminus \{\beta\}, V, C \rangle, \mathcal{B})$ **do**
if β is `assert` **then**
if β evaluates deterministically **then**
if $\beta(s'')$ **then yield** $\langle s', \bar{a}, V', C' \rangle$ ▷ Yield only if $\beta(s'')$ is true
else yield $\langle s', \bar{a}, V', C' \cup \{\beta\} \rangle$ ▷ Defer β check to CSP
else if β is `commit` **then**
if Solve-CSP(V', C') returns an assignment A **then**
yield $\langle s'_{V \setminus A}, \bar{a}_{V \setminus A}, \emptyset, \emptyset \rangle$; **break** ▷ Yield only the first plan trace
else if β is `achieve` **then**
for $b = \langle L', M', R' \rangle \in \mathcal{B} \mid goal(b) = \beta$ **do**
yield from Crow$(s', \langle L', M', R', V', C' \rangle, \mathcal{B})$
else if β is `do` **then yield** $\langle s', \bar{a} \oplus \{\beta\}, V', C' \rangle$
else if β is `bind` **then return** $\langle s', \bar{a}, V' \cup \{var(\beta)\}, C' \cup \{body(\beta)\} \rangle$
else if β is `effect` **then return** $\langle update(s', \beta), \bar{a}, V', C' \rangle$

The function Crow defined above is not a good control structure, as it effectively does a left-branching depth-first traversal of the space of plans. To make it practical, we wrap it in an iterative-deepening depth-first search (IDDFS), which searches with an increasing depth limit until a sound plan is found.

Theorem 12.3.1. *Suppose there exists a sound plan that can be generated by the behavior specification $\mathcal{B}$ at a finite depth of nested* `achieve` *statements. Then the IDDFS* Crow *will always terminate and return a sound plan.*

(Proof in appendix M.1.1). While the IDDFS Crow algorithm is easy to implement and comprehend, other implementations such as breadth-first search or A-Star search with heuristics, would also be possible.

12.3.1 Problem Hardness and Crow Runtime

In discrete-space planning without `bind` statements, based on languages like PDDL, it is typical to be completely general and put all `achieve` statements into the `promotable` section, which can result in poor worst-case runtimes. However, if we know that there are some subgoals whose achievement is independent of others, then we can put all and only the `achieve` statements that could depend on each other into the promotable section, and for the other, independent subgoals, commit after achieving each one. In such cases, an upper bound on the problem complexity is related to the size of the largest $\langle L, M, R\rangle$ tuple during the refinement process. Here, we restate the problem hardness results from STRIPS planning [Bylander, 1994] in the language of cdl.

Theorem 12.3.2 (Regression Width (Chapter 4))**.** *Consider the refinement process applied to an initial program P_0 and an initial state s_0, given a set of behaviors $\mathcal{B}$. If there is a sound plan that can be refined by allowing maximally K* `achieve` *statements in $M + R$ of any partially refined plan traces $\langle L, M, R\rangle$, the complexity of finding a sound plan is upper-bounded by $N^{O(K)}$ where N is the number of entities in the environment, and K is called the "width" of the planning problem.*

Proof sketch. For all possible programs p, let $subgoals(p)$ be the set of `achieve` statements inside p. No matter the order of elements in p, all we care about is finding a feasible plan that simultaneously achieves all subgoals in $subgoals(p)$, and there are $N^{O(K)}$ possible subsets. Thus, it can be solved in at most $N^{O(K)}$ time.

This result gives us a fine-grained insight into a useful class of planning problems in terms of their hardness, which essentially depends on the total number of subgoals that accumulate in the program, usually in the "middle" section. In Chapter 4, we have proved that, for many practical problems, K is a small constant usually independent of the number of entities in the environment. For practitioners, this suggests an alternative search algorithm for implementing a cdl interpreter, which is an iterative-widening algorithm [Lipovetzky and Geffner, 2012], where, instead of iterating over the depth of the refinement process, we iterate over the maximal number of `achieve` statements allowed to accumulate during the refinement process. In this case, the iterative-widening (Crow-IW) algorithm will be able to solve the problem in time bounded by $N^{O(K)}$. This property of width also helps us gain insights into the CSP solving hardness in Crow. Essentially, if we hypothesize that the amount of time needed for solving continuous CSPs is exponential in the number of variables, we would obtain a complexity of Crow on generic hybrid decision-making problems which is a polynomial whose exponent is linear in the program width and the number of variables accumulated in CSPs (due to non-`commit`). This hypothesis could be difficult to satisfy in problems where generating values that satisfy the constraints (e.g. by

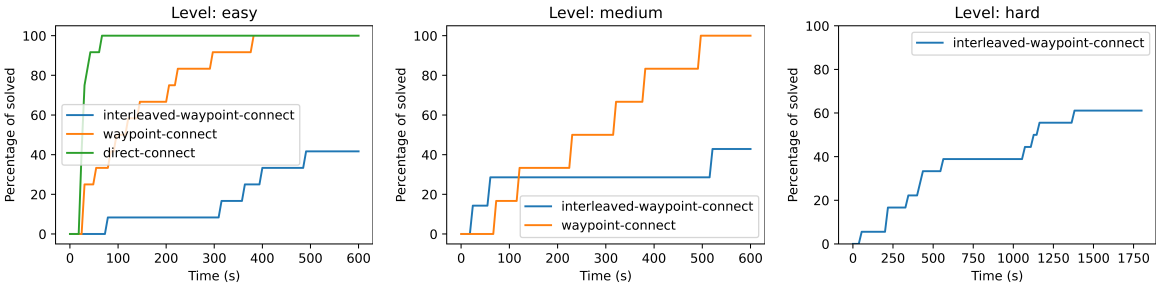


Figure 12-3: **Results on NAMO problems.** Percentage of successful executions as a function of wall-clock seconds; each plot corresponds to a NAMO problem instance, and each line-color corresponds to a behavior program.

sampling) is hard.

12.3.2 Experiments in NAMO

In the introduction, we used the NAMO problem to illustrate the flexibility of CDL for expressing different spaces of possible plans, and argued that this flexibility is useful because a general-purpose planning algorithm can take advantage of reduced plan spaces without requiring special-purpose algorithms to be implemented. Here, we explore the effectiveness of the CDL and our interpreter on those three CDL behavior specifications: the *direct-connect* approach which first plans the path from the initial position to the goal and then clears all obstacles it hits; the *waypoint-connect* approach which searches for a set of intermediate waypoints, and plans for the whole path by clearing all obstacles between each consecutive waypoint in a stagewise manner; and the *interleaved-waypoint-connect* approach which considers all possible orders for the movements and obstacle-clearing. Intuitively, these three algorithms are ordered in a way that the later algorithms can solve a large set of problems than previous ones.

We consider the three example NAMO problems presented in Fig. 12-1 and apply the three behavior programs to these problems, and show the results in Figure 12-3. Since there is substantial nondeterminism in the algorithm, we run CROW on each behavior program 10 times for each problem. The results match our intuition of the benefits of providing a behavior specification appropriate to the problem class. Overall, if we apply a more advanced strategy for solving "simpler" problems (e.g., using the waypoint-connect algorithm for the easy-level NAMO tasks), due to the additional branching factors it considers (for example, the direct-connect approach does not consider waypoints while the waypoint-connect algorithm will try sampling different waypoints), the overall runtime is longer than for the methods specifically tailored towards the scenario, although given a sufficient amount of time, they are capable of solving the problem.

Importantly, we did not have to write any new code to get these results: the general Crow algorithm running on the simple alternative strategies in Fig. 12-1 is all that was necessary, and it would be easy to further explore a large the trade-off space of behavior specifications, generality, and efficiency.

12.4 Related Work

cdl builds on a series of insights from hierarchical planning [Alford *et al.*, 2016], subgoal serialization [Lipovetzky and Geffner, 2012, ; see also Chapter 4], and constraint satisfaction-based hybrid discrete-continuous planning [Garrett *et al.*, 2018, 2021]. Its semantics allow for the specification of a large body of behavior types. For example, behavior trees [Colledanchise and Ögren, 2018] can be described by cdl behaviors without effect sections. Preconditions of rules can be written as assert statements at the top of the body. A classical STRIPS operator definition [Fikes and Nilsson, 1971] is equivalent to a cdl behavior with all preconditions written in its promotable section (but note that the original STRIPS *implementation* had all preconditions `unordered` but not `promotable`, making it incomplete). Furthermore, any atomic STRIPS effect statements [Lifschitz, 1987] can also be written as assignment statements in cdl. Therefore, it is one of the behavior types with the weakest commitments. In this case, without any further control structure, the interpreter Crow is sound and complete but very inefficient. This could be improved by adding heuristics [Hoffmann and Nebel, 2001; Helmert, 2006]. The most important benefit of cdl is that when programmers know what can be serialized, they can use cdl to declare those commits to make search efficient.

The most closely-related planning frameworks are the hierarchical task and goal networks [Erol *et al.*, 1994b; Alford *et al.*, 2016], where an operator is equivalent to a cdl behavior with an ordered promotable action body containing preconditions: they are always ordered but interleavable. However, being a uniform approach that semantically always allows for interleaved execution of subgoal routines, they support neither serialization nor commits. All these behavior specifications can be implemented in cdl, as well as combined flexibly to address non-homogenous problems.

The idea of explicitly separating problem-solving into generators and tests dates back Newell and Simon [Newell *et al.*, 1972], and was more recently applied to robot programming and planning starting with GOLOG [Levesque *et al.*, 1997], its variants [De Giacomo *et al.*, 2000, 2009], and Planning with Loops [Levesque, 2005]. However, their generators are specified in a generic programming language where interleaved behaviors, commits, and constraints need to be programmed manually. cdl has the same spirit as these algorithms but is built on insights from hierarchical planning, subgoal serialization, and constraint-based hybrid planning to define a set of language supports.

12.5 Discussion

We have discussed at length the strengths of CDL and CROW as a framework for *planning*; but it also exposes a set of opportunities for *learning*. In Chapter 11, we explored how to learn transition models and policies for hybrid domains, and in Chapter 5, we investigated learning samplers for TAMP systems [Mendez-Mendez *et al.*, 2023; Yang *et al.*, 2023b; Ha *et al.*, 2022]. While these components have so far been studied in isolation, CDL offers a unifying formalism for integrating learned partial models, policies, and samplers as compositional behavior rules. Moreover, we have demonstrated the critical role of ordering, assertions, and commitment in making planning efficient. This reveals a new learning opportunity: to automatically discover and incorporate such inference-strategy elements. Looking forward, we hope to bring together all of these directions: transition models, policies, samplers, and inference strategies, into a single integrated framework.

Chapter 13

Epilogue: All You Need is Everything

We have presented a general framework for learning, reasoning, and planning that applies to a variety of domains and tasks. The key principle is to learn an abstract and compositional model of the world and reason with it. Based on the foundation of a neuro-symbolic concept representation, our approach can continuously learn concepts from data streams in a data-efficient manner and compose its learned representations in infinitely many ways to reason about new tasks, including previously unseen ones.

Our ultimate goal is to combine models, learning and reasoning algorithms, as well as available data, to build a system that can exhibit intelligent goal-driven behaviors. Looking forward, this dissertation outlines a concrete proposal for this problem: building intelligence through reasoning over learned compositional abstractions — entities, concepts, and the relations among them. Rather than encoding experiences pixel by pixel or text word by word (the "analogical" representations), we propose to structure them into conceptual representations governed by logic, probability, causality, and physical principles (the "symbolic" representations). Following this level of analysis from David Marr, at the algorithmic level, the neuro-symbolic concept learning and reasoning framework allows all these representations and reasoning algorithms to be integrated within a single framework.

At the implementational level, I argue that, despite their conceptual differences, computationally, manually designed algorithms and learnable neural networks can all be viewed as (parallel) circuits that perform computation. Typically, a neural network contains only simple feed-forward computation patterns based primarily on matrix multiplications, whereas algorithmic circuits are more flexible: they may contain recurrent or recursive computations and more sophisticated memory access patterns (such as tree search).

Therefore, our real goal should be to develop representations and algorithms that leverage the structures of the problem to be both efficient and effective under rational resource constraints, including data collection effort, learning cost, inference cost, and inference efficiency. The central bet suggested by the promising results in this dissertation is that

a truly general-purpose intelligent machine, capable of processing various modalities of observations and continually learning, adapting, and acting, should likely take the form of a computation that goes beyond any single existing neural network architecture, such as transformers.

The first step of our research program, as the primary proposal in this dissertation, is to deeply integrate existing neural network architectures and algorithms for perception, language understanding, logical reasoning, constraint satisfaction solving, and planning. Given multi-modal observations, including perceptual signals and language instructions, we need a "problem formulation" module, neural (e.g., large language models) or symbolic (e.g., a fixed set of rules for the target domain specified by the user), that can generate a problem formulation. The problem formulation includes an abstract representation of the state, the possible actions the machine can take, and human intention. These abstract representations can be obtained by composing perception modules, neural (e.g., large vision models), or symbolic (e.g., analysis by synthesis with human-built physics engines). The response can then be computed with a reasoning algorithm, neural (e.g., Diffusion-CCSP; see Chapter 5), or symbolic (e.g., planning algorithms such as CROW; see Chapter 12). This thesis proposes a particular composition: large language models for generating problem formulations, modality-specific neural modules for perception, and a composition of symbolic search and neural generative-model-based probabilistic inference for continuous value optimization. This integrates some of the best ideas from AI, machine learning, computer vision, natural language processing, and robotics into a holistic system with strong learning and generalization capabilities.

Several future directions remain to be explored within this framework. One is the formulation of problems at test time (e.g., defining the state representation and transition rules). In the work presented in this dissertation, such as Chapter 11, we assumed that for a given training domain, the state representation and transition rules (i.e., the world model) are either given or learned, but remain fixed at test time. Another important direction is memory and adaptation: the machine must be able to maintain memory about space, objects, and other agents, including their beliefs, goals, and preferences. Closely tied to the problem of memory updating is the capability to handle partial observability and to adapt to uncertain factors in the world model (e.g., the friction of an object cannot always be pre-determined from vision alone).

Another important open question concerns the lifetime of a robot. When constructing a robot in the factory, we must ask what models should be built in from the start, whether low-level physical dynamics, commonsense knowledge about the everyday world, or high-level abstractions distilled from large corpora such as Wikipedia. Once deployed in a household or workplace, the robot must continue to adapt: learning from its own interactions, from

feedback on human preferences, and from the evolving stream of online data. This raises the challenge of continual learning and preference learning: how to preserve core competencies while expanding its repertoire. Finally, at performance time, the robot must not only act autonomously but also collaborate effectively with humans, both in information-seeking (e.g., asking for clarification or information) and in joint action.

In summary, this dissertation makes the case for a unified framework for learning, reasoning, and planning: one that treats perception, language, and action not as isolated modules, but as interconnected components of a full computational system. By grounding intelligence in compositional abstractions and by integrating neural and symbolic methods under rational resource constraints, our long-term vision is to build systems that can continually expand their *concept* repertoire, communicate and collaborate with humans, and act effectively in open-ended environments.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 14

Encore: Everything is One Thing

I would like to note that, in the development of all the theories, algorithms, and systems presented in this dissertation, the single shared research idea is *unification*. In Part I, we introduced the unification of first-order and temporal logic, with neural network architectures (Chapters 2 and 3). Then, we introduced the unification of fine-grained planning problem complexity with neural network circuit complexity (Chapter 4). In Chapter 5, we presented a unification of constraint satisfaction solving with compositional diffusion models. These new algorithms for logical reasoning and continuous constraint satisfaction solving lay a critical foundation for real-world reasoning and planning tasks, such as in vision and robotics.

Moving on to the vision-language domain in Part II, we focused on the unification of visual concept learning with language acquisition. In particular, we started with a framework in Chapter 6 that is a unification of visual concept learning with reasoning: the acquisition of new words in language introduces new visual concepts to be learned (e.g., object properties and relations). Answering natural language questions is done by composing learned visual concepts with a programmatic structure. Chapter 7 introduces the unification of concept learning with metaconcept learning (learning the relationships among concepts). Chapter 8 presents a unification of reasoning across domains and modalities. Finally, in Chapter 9 we studied the unification of neuro-symbolic concept representation (the abstract and compositional meaning of words) with language syntax (the compositional rules of words in languages).

In Part III, we brought unification into the realm of decision-making and robotics, starting with the unification of contact modeling and reasoning with hybrid discrete-continuous planning, focusing on contact as the key discrete structure. In Chapter 10, we present a general framework for learning and composing manipulation strategies. Chapter 11 introduces the unification of planning domain programming with learning, and the unification of symbolic structures in domain transition models with neural computation graphs, which

encode sparsity and locality structures of the environment. Finally, in Chapter 12, we present a unification of declarative planning with imperative policy execution, allowing agents to move fluidly between planning and policy following, as well as discrete search and continuous constraint satisfaction solving.

Throughout the history of AI research, debates have persisted around different "algorithmic strategies" for intelligence: symbolism versus connectionism, model-based planning versus model-free learning, and top-down language guidance versus bottom-up perceptual abstraction. Many research (sub-)communities have committed to a single mechanism, pushing it to its extremes. Yet, time and again, these debates have revealed themselves as trade-offs. For example, one approach may excel in test-time efficiency while another reduces data requirements. Usually, the answer to which is "better" often collapses to the unsatisfying but practical response: *it depends on the problem*. A natural scientific goal, then, is to identify criteria that let us predict in advance which strategy will be most effective for a given setting: for example, inspecting features of a decision-making problem to decide when model-based planning is preferable.

But even this view turns out to be incomplete. As the Navigation Among Movable Obstacles (NAMO) example in Chapter 12 illustrates, different solution strategies often need to be combined to solve a single instance, and even within a single problem class, different instances may call for different methods. The evidence presented in this dissertation, spanning logic, constraint solving, concept learning, language acquisition, and robotic manipulation, all suggest that *these seemingly disparate strategies are not fundamentally dichotomic*. In some cases, they are provably equivalent in certain aspects (e.g., in terms of expressiveness, see Chapter 2). In others, they lie along a continuous spectrum (e.g., the imperative–declarative hybrids that interpolate between policies and models; see Chapter 12). Taken together, this points toward the possibility that all of these techniques are instantiations of a single, compact, underlying principle.

However, we are still far away from that "grand unified theory of intelligence," in at least three aspects. First, all the unifications presented in this dissertation are somewhat "pairwise," in the sense that they relate two concepts or techniques together. We still need to move upward in the abstraction hierarchy to develop frameworks that unify larger families of techniques simultaneously (for instance, continuous constraint satisfaction and sequential planning are treated completely differently in Chapter 12). Second, our current scope omits many important learning and reasoning capacities, such as abstract reasoning (e.g., in mathematics or literature), preference and utility learning, and the inference of other agents' beliefs, desires, and intentions. Third, the "path" to intelligence is inherently conditioned on resource constraints, such as data availability, training cost, and inference cost. Humans and machines differ dramatically in memory capacity, lifetimes, and modes of interaction

with the world, and therefore the optimal way of achieving the same level of intelligence may diverge substantially. A comprehensive theory must therefore not only unify different learning and reasoning methods but also articulate how intelligence can be realized under different resource conditions.

What might such a unified theory look like in practice? At the algorithmic level, one possibility is that all important classes of algorithms — search, optimization, inference, planning, learning — can be expressed compactly and efficiently within a single representational framework, so that they can be seamlessly integrated rather than treated as isolated toolkits. At the computational level, the theory could articulate systematic ways to measure the inherent complexity of problems: how much data is needed to learn a new concept, how structural biases or built-in models accelerate learning, and, as a corollary, how different resources (memory, lifetime, interaction bandwidth) change the feasible trajectories toward intelligence. Such a theory could potentially explain aspects of learning and reasoning in existing intelligent systems: humans and animals, and also help us predict and engineer the possible developmental pathways for machine intelligence.

Nevertheless, I am inclined to believe that a central theme of the pursuit of understanding intelligence and building intelligence is the pursuit of unification in a "standard model." It remains uncertain what form such a unified theory might take, or whether it can ever be fully captured, but the possibility itself offers a compelling direction for both scientific inquiry and system design.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, pages 265–283, 2016.

Omri Abend, Tom Kwiatkowski, Nathaniel J Smith, Sharon Goldwater, and Mark Steedman. Bootstrapping Language Acquisition. *Cognition*, 164:116–143, 2017.

Bernardo Aceituno-Cabezas and Alberto Rodriguez. A Global Quasi-Dynamic Model for Contact-Trajectory Optimization in Manipulation. In *RSS*, 2020.

Panos Achlioptas, Ahmed Abdelreheem, Fei Xia, Mohamed Elhoseiny, and Leonidas Guibas. ReferIt3D: Neural Listeners for Fine-Grained 3D Object Identification in Real-World Scenes. In *ECCV*, 2020.

Alper Ahmetoglu, Erhan Oztop, and Emre Ugur. Symbolic Manipulation Planning with Discovered Object and Relational Predicates. *RA-L*, 2025.

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as I Can and Not as I Say: Grounding Language in Robotic Affordances. In *arXiv:2204.01691*, 2022.

Ekin Akyürek, Afra Feyza Akyürek, and Jacob Andreas. Learning to Recombine and Resample Data for Compositional Generalization. In *ICLR*, 2021.

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. Flamingo: A Visual Language Model for Few-Shot Learning. In *NeurIPS*, 2022.

Jacopo Aleotti and Stefano Caselli. Manipulation Planning of Similar Objects by Part Correspondence. In *ICRA*, 2011.

Ron Alford, Vikas Shivashankar, Mark Roberts, Jeremy Frank, and David W Aha. Hierarchical Planning: Relating Task and Goal Decomposition with Task Sharing. In *IJCAI*, 2016.

Peter Anderson, Xiaodong He, Chris Buehler, Damien Teney, Mark Johnson, Stephen Gould, and Lei Zhang. Bottom-Up and Top-Down Attention for Image Captioning and Visual Question Answering. In *CVPR*, 2018.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Learning to Compose Neural Networks for Question Answering. In *NAACL-HLT*, 2016.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural Module Networks. In *CVPR*, 2016.

Jacob Andreas, Dan Klein, and Sergey Levine. Modular Multitask Reinforcement Learning with Policy Sketches. In *ICML*, 2017.

Jacob Andreas. Good-Enough Compositional Data Augmentation. In *ACL*, 2020.

Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C Lawrence Zitnick, and Devi Parikh. VQA: Visual Question Answering. In *ICCV*, 2015.

Yoav Artzi and Luke Zettlemoyer. Weakly Supervised Learning of Semantic Parsers for Mapping Instructions to Actions. *TACL*, 2013.

K Somani Arun, Thomas S Huang, and Steven D Blostein. Least-Squares Fitting of Two 3-D Point Sets. *T-PAMI*, 9(5):698–700, 1987.

Haruo Asada and Michael Brady. The Curvature Primal Sketch. *T-PAMI*, PAMI-8(1):2–14, 1986.

Masataro Asai and Christian Muise. Learning Neural-Symbolic Descriptive Planning Models via Cube-Space Priors: The Voyage Home (To STRIPS). *arXiv:2004.12850*, 2020.

Anas Awadalla, Irena Gao, Josh Gardner, Jack Hessel, Yusuf Hanafy, Wanrong Zhu, Kalyani Marathe, Yonatan Bitton, Samir Gadre, Shiori Sagawa, et al. OpenFlamingo: An Open-Source Framework for Training Large Autoregressive Vision-Language Models. *arXiv:2308.01390*, 2023.

Waiss Azizian and Marc Lelarge. Expressive Power of Invariant and Equivariant Graph Neural Networks. In *ICLR*, 2021.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. In *ICLR*, 2015.

Renée Baillargeon and Gerald F DeJong. Explanation-Based Learning in Infancy. *Psychonomic Bulletin & Review*, 24:1511–1526, 2017.

Michael Bain and Claude Sammut. A Framework for Behavioural Cloning. *Machine Intelligence*, pages 103–129, 1995.

Chris L Baker, Julian Jara-Ettinger, Rebecca Saxe, and Joshua B Tenenbaum. Rational Quantitative Attribution of Beliefs, Desires and Percepts in Human Mentalizing. *Nature Human Behaviour*, 1(4):1–10, 2017.

Pablo Barceló, Egor V Kostylev, Mikael Monet, Jorge Pérez, Juan Reutter, and Juan Pablo Silva. The Logical Expressiveness of Graph Neural Networks. In *ICLR*, 2020.

Marco Baroni. Linguistic Generalization and Compositionality in Modern Artificial Neural Networks. *Philosophical Transactions of the Royal Society B*, 375(1791):20190307, 2020.

Anthony Barrett and Daniel S Weld. Characterizing Subgoal Interactions for Planning. In *IJCAI*, 1993.

David Barrett, Felix Hill, Adam Santoro, Ari Morcos, and Timothy Lillicrap. Measuring Abstract Reasoning in Neural Networks. In *ICML*, 2018.

Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational Inductive Biases, Deep Learning, and Graph Networks. *arXiv:1806.01261*, 2018.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum Learning. In *ICML*, 2009.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting Is Programming: A Query Language for Large Language Models. In *PLDI*, 2023.

Ondrej Biza, Skye Thompson, Kishore Reddy Pagidi, Abhinav Kumar, Elise van der Pol, Robin Walters, Thomas Kipf, Jan-Willem van de Meent, Lawson L. S. Wong, and Robert Platt. One-Shot Imitation Learning via Interaction Warping. In *CORL*, 2023.

Paul Bloom. *How Children Learn the Meanings of Words*. The MIT Press, 2000.

Andreea Bobu, Chris Paxton, Wei Yang, Balakumar Sundaralingam, Yu-Wei Chao, Maya Cakmak, and Dieter Fox. Learning Perceptual Concepts by Bootstrapping From Human Queries. *RA-L*, 2022.

Kurt D. Bollacker, Colin Evans, Praveen K. Paritosh, Tim Sturge, and Jamie Taylor. Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge. In Jason Tsong-Li Wang, editor, *SIGMOD*, 2008.

Blai Bonet and Héctor Geffner. Planning as Heuristic Search. *AIJ*, 129(1-2):5–33, 2001.

Blai Bonet and Hector Geffner. Learning First-Order Symbolic Representations for Planning From the Structure of the State Space. In *ECAI*, 2020.

Antoine Bordes, Nicolas Usunier, Alberto García-Durán, Jason Weston, and Oksana Yakhnenko. Translating Embeddings for Modeling Multi-Relational Data. In *NeurIPS*, 2013.

Sascha Brandi, Oliver Kroemer, and Jan Peters. Generalizing Pouring Actions Between Objects Using Warped Parameters. In *Humanoids*, 2014.

William Brendel, Alan Fern, and Sinisa Todorovic. Probabilistic Event Logic for Interval-Based Event Recognition. In *CVPR*, 2011.

Joan Bresnan, Ash Asudeh, Ida Toivonen, and Stephen Wechsler. *Lexical-Functional Syntax*. John Wiley & Sons, 2016.

Blandine Bril, Robert Rein, Tetsushi Nonaka, Francis Wenban-Smith, and Gilles Dietrich. The Role of Expertise in Tool Use: Skill Differences in Functional Action Adaptations to Task Constraints. *Journal of Experimental Psychology: Human Perception and Performance*, 36(4):825, 2010.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language Models Are Few-Shot Learners. In *NeurIPS*, 2020.

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral Networks and Locally Connected Networks on Graphs. In *ICLR*, 2014.

Davide Buffelli, Pietro Liò, and Fabio Vandin. SizeShiftReg: A Regularization Method for Improving Size-Generalization in Graph Neural Networks. In *NeurIPS*, 2022.

Rudy R Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. In *ICLR*, 2018.

Tom Bylander. The Computational Complexity of Propositional STRIPS Planning. *AIJ*, 69(1-2):165–204, 1994.

Jin-Yi Cai, Martin Fürer, and Neil Immerman. An Optimal Lower Bound on the Number of Variables for Graph Identification. *Combinatorica*, 12(4):389–410, 1992.

Jonathon Cai, Richard Shin, and Dawn Song. Making Neural Programming Architectures Generalize via Recursion. In *ICLR*, 2017.

Alberto Camacho, Jorge Baier, Christian Muise, and Sheila McIlraith. Finite LTL Synthesis as Planning. In *ICASP*, 2018.

Andres Campero, Aldo Pareja, Tim Klinger, Joshua B. Tenenbaum, and Sebastian Riedel. Logical Rule Induction and Theory Learning Using Neural Theorem Proving. *arXiv:1809.02193*, 2018.

John Canny. A Computational Approach to Edge Detection. *T-PAMI*, 8(6):679–698, 1986.

Susan Carey and E. Bartlett. Acquiring a Single New Word. *Proceedings of the Stanford Child Language Conference*, 15:17–29, 01 1978.

Thomas Carta, Clément Romac, Thomas Wolf, Sylvain Lamprier, Olivier Sigaud, and Pierre-Yves Oudeyer. Grounding Large Language Models in Interactive Environments with Online Reinforcement Learning. In *ICML*, 2023.

Angel X Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. ShapeNet: An Information-Rich 3D Model Repository. *arXiv:1512.03012*, 2015.

Hubie Chen and Omer Giménez. Act Local, Think Global: Width Notions for Tractable Planning. In *ICAPS*, 2007.

Kan Chen, Jiang Wang, Liang-Chieh Chen, Haoyuan Gao, Wei Xu, and Ram Nevatia. ABC-CNN: An Attention Based Convolutional Neural Network for Visual Question Answering. *arXiv:1511.05960*, 2015.

Xinyun Chen, Chang Liu, and Dawn Song. Towards Synthesizing Complex Programs From Input-Output Examples. In *ICLR*, 2018.

Jiajun Chen, Huarui He, Feng Wu, and Jie Wang. Topology-Aware Correlations Between Relations for Inductive Link Prediction in Knowledge Graphs. In *AAAI*, 2021.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision Transformer: Reinforcement Learning via Sequence Modeling. In *NeurIPS*, 2021.

Wenhu Chen, Zhe Gan, Linjie Li, Yu Cheng, William Wang, and Jingjing Liu. Meta Module Network for Compositional Visual Reasoning. In *WACV*, 2021.

Zhenfang Chen, Jiayuan Mao, Jiajun Wu, Kwan-Yee Kenneth Wong, Joshua B Tenenbaum, and Chuang Gan. Grounding Physical Concepts of Objects and Events Through Dynamic Visual Reasoning. In *ICLR*, 2021.

Yongchao Chen, Jacob Arkin, Yang Zhang, Nicholas Roy, and Chuchu Fan. AutoTAMP: Autoregressive Task and Motion Planning with LLMs as Translators and Checkers. In *ICRA*, 2024.

Kewei Cheng, Ziqing Yang, Ming Zhang, and Yizhou Sun. UniKER: A Unified Framework for Combining Embedding and Definite Horn Rule Reasoning for Knowledge Graph Inference. In *EMNLP*, nov 2021.

Xianyi Cheng, Eric Huang, Yifan Hou, and Matthew T Mason. Contact Mode Guided Sampling-Based Planning for Quasistatic Dexterous Manipulation in 2D. In *ICRA*, 2021.

Xianyi Cheng, Eric Huang, Yifan Hou, and Matthew T Mason. Contact Mode Guided Motion Planning for Quasidynamic Dexterous Manipulation in 3D. In *ICRA*, 2022.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. Binding Language Models in Symbolic Languages. In *ICLR*, 2023.

Maxime Chevalier-Boisvert, Dzmitry Bahdanau, Salem Lahlou, Lucas Willems, Chitwan Saharia, Thien Huu Nguyen, and Yoshua Bengio. BabyAI: First Steps Towards Grounded Language Learning with a Human in the Loop. In *ICLR*, 2019.

Cheng Chi, Siyuan Feng, Yilun Du, Zhenjia Xu, Eric Cousineau, Benjamin Burchfiel, and Shuran Song. Diffusion Policy: Visuomotor Policy Learning via Action Diffusion. In *RSS*, 2023.

Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *KDD*, 2019.

Silvia Chiappa, Sébastien Racaniere, Daan Wierstra, and Shakir Mohamed. Recurrent Environment Simulators. In *ICLR*, 2017.

Rohan Chitnis, Tom Silver, Joshua B Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Learning Neuro-Symbolic Relational Transition Models for Bilevel Planning. In *IROS*, 2022.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation. In *EMNLP*, 2014.

Jaemin Cho, Jie Lei, Hao Tan, and Mohit Bansal. Unifying Vision-and-Language Tasks via Text Generation. In *ICML*, 2021.

Glen Chou, Necmiye Ozay, and Dmitry Berenson. Explaining Multi-Stage Tasks by Learning Temporal Logic Formulas From Suboptimal Demonstrations. In *RSS*, 2020.

Grzegorz Chrupała, Akos Kádár, and Afra Alishahi. Learning Language Through Pictures. In *ACL*, 2015.

John Cocke. *Programming Languages and Their Compilers: Preliminary Notes*. New York University, 1969.

Michele Colledanchise and Petter Ögren. *Behavior Trees in Robotics and AI: An Introduction*. CRC Press, 2018.

Joseph Culberson. Sokoban Is PSPACE-complete. Technical report, University of Alberta, 1997.

Aidan Curtis, Xiaolin Fang, Leslie Pack Kaelbling, Tomás Lozano-Pérez, and Caelan Reed Garrett. Long-Horizon Manipulation of Unknown Objects via Task and Motion Planning with Estimated Affordances. In *ICRA*, 2022.

Angela Dai, Angel X Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Nießner. ScanNet: Richly-Annotated 3D Reconstructions of Indoor Scenes. In *CVPR*, 2017.

Wang-Zhou Dai, Qiu-Ling Xu, Yang Yu, and Zhi-Hua Zhou. Tunneling Neural Perception and Logic Reasoning Through Abductive Learning. *arXiv:1802.01173*, 2018.

Murtaza Dalal, Tarun Chiruvolu, Devendra Chaplot, and Ruslan Salakhutdinov. Plan-Seq-Learn: Language Model Guided RL for Solving Long Horizon Robotics Tasks. In *ICLR*, 2024.

Neil T Dantam, Zachary K Kingston, Swarat Chaudhuri, and Lydia E Kavraki. Incremental Task and Motion Planning: A Constraint-Based Approach. In *RSS*, 2016.

Neil T Dantam, Zachary K Kingston, Swarat Chaudhuri, and Lydia E Kavraki. An Incremental Constraint-Based Framework for Task and Motion Planning. *IJRR*, 37(10):1134–1151, 2018.

Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and Expressive Power of Logic Programming. *ACM CSUR*, 33:374–425, 2001.

Srijan Das, Rui Dai, Michal Koperski, Luca Minciullo, Lorenzo Garattoni, Francois Bremond, and Gianpiero Francesca. Toyota Smarthome: Real-World Activities of Daily Living. In *ICCV*, 2019.

Peter Dayan and Geoffrey E Hinton. Feudal Reinforcement Learning. In *NeurIPS*, 1992.

Giuseppe De Giacomo, Yves Lespérance, and Hector J Levesque. ConGolog: A Concurrent Programming Language Based on the Situation Calculus. *AIJ*, 121(1-2):109–169, 2000.

Giuseppe De Giacomo, Yves Lespérance, Hector J Levesque, and Sebastian Sardina. IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents. *Multi-Agent Programming: Languages, Tools and Applications*, pages 31–72, 2009.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *NeurIPS*, 2016.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal Transformers. In *ICLR*, 2019.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*, 2009.

Zhiwei Deng, Arash Vahdat, Hexiang Hu, and Greg Mori. Structure Inference Machines: Recurrent Neural Networks for Analyzing Relations in Group Activity Recognition. In *CVPR*, 2016.

Misha Denil, Sergio Gómez Colmenarejo, Serkan Cabi, David Saxton, and Nando de Freitas. Programmable Agents. *arXiv:1706.06383*, 2017.

Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2D Knowledge Graph Embeddings. In *AAAI*, 2018.

Coline Devin, Pieter Abbeel, Trevor Darrell, and Sergey Levine. Deep Object-Centric Representations for Generalizable Robot Learning. In *ICRA*, 2018.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural Program Learning Under Noisy I/O. In *ICML*, 2017.

Surís Dídac, Sachit Menon, and Carl Vondrick. ViperGPT: Visual Inference via Python Execution for Reasoning. In *ICCV*, 2023.

Carlos Diuk, Andre Cohen, and Michael L Littman. An Object-Oriented Representation for Efficient Reinforcement Learning. In *ICML*, 2008.

Mehmet R Dogar and Siddhartha S Srinivasa. A Planning Framework for Non-Prehensile Manipulation Under Clutter and Uncertainty. *Autonomous Robots*, 33:217–236, 2012.

David Dohan, Winnie Xu, Aitor Lewkowycz, Jacob Austin, David Bieber, Raphael Gontijo Lopes, Yuhuai Wu, Henryk Michalewski, Rif A Saurous, Jascha Sohl-Dickstein, et al. Language Model Cascades. In *ICML Workshop*, 2022.

Jeffrey Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-Term Recurrent Convolutional Networks for Visual Recognition and Description. In *CVPR*, 2015.

Li Dong and Mirella Lapata. Language to Logical Form with Neural Attention. In *ACL*, 2016.

Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. Neural Logic Machines. In *ICLR*, 2019.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *ICLR*, 2021.

Laura Downs, Anthony Francis, Nate Koenig, Brandon Kinman, Ryan Hickman, Krista Reymann, Thomas B McHugh, and Vincent Vanhoucke. Google Scanned Objects: A High-Quality Dataset of 3D Scanned Household Items. In *ICRA*, 2022.

Dominik Drexler, Jendrik Seipp, and Hector Geffner. Learning Sketches for Decomposing Planning Problems Into Subproblems of Bounded Width. In *ICAPS*, 2022.

Danny Driess, Ozgur Oguz, Jung-Su Ha, and Marc Toussaint. Deep Visual Heuristics: Learning Feasibility of Mixed-Integer Programs for Manipulation Planning. In *ICRA*, 2020.

Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. PaLM-E: An Embodied Multimodal Language Model. *arXiv:2303.03378*, 2023.

Andrew Drozdov, Pat Verga, Mohit Yadav, Mohit Iyyer, and Andrew McCallum. Unsupervised Latent Tree Induction with Deep Inside-Outside Recursive Autoencoders. In *NAACL-HLT*, 2019.

Yilun Du, Shuang Li, and Igor Mordatch. Compositional Visual Generation with Energy Based Models. In *NeurIPS*, 2020.

Yilun Du, Conor Durkan, Robin Strudel, Joshua B Tenenbaum, Sander Dieleman, Rob Fergus, Jascha Sohl-Dickstein, Arnaud Doucet, and Will Grathwohl. Reduce, Reuse, Recycle: Compositional Generation with Energy-Based Diffusion Models and McMc. In *ICML*, 2023.

Yilun Du, Mengjiao Yang, Pete Florence, Fei Xia, Ayzaan Wahid, Brian Ichter, Pierre Sermanet, Tianhe Yu, Pieter Abbeel, Joshua B Tenenbaum, et al. Video Language Planning. In *ICLR*, 2024.

Herbert Edelsbrunner, David Kirkpatrick, and Raimund Seidel. On the Shape of a Set of Points in the Plane. *IEEE Transactions on Information Theory*, 29(4):551–559, 1983.

Jens Egeblad, Benny K Nielsen, and Allan Odgaard. Fast Neighborhood Search for Two- And Three-Dimensional Nesting Problems. *European Journal of Operational Research*, 183(3):1249–1266, 2007.

Kiana Ehsani, Shubham Tulsiani, Saurabh Gupta, Ali Farhadi, and Abhinav Gupta. Use the Force, Luke! Learning to Predict Physical Forces by Simulating Effects. In *CVPR*, 2020.

Y. Eldar, M. Lindenbaum, M. Porat, and Y.Y. Zeevi. The Farthest Point Strategy for Progressive Image Sampling. *IP*, 6(9):1305–1315, 1997.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. DreamCoder: Growing Generalizable, Interpretable Knowledge with Wake-Sleep Bayesian Program Learning. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 381(2251):20220050, 2023.

Mark Endo, Joy Hsu, Jiaman Li, and Jiajun Wu. Motion Question Answering via Modular Motion Programs. In *ICML*, 2023.

Kutluhan Erol, James Hendler, and Dana S Nau. HTN Planning: Complexity and Expressivity. In *AAAI*, 1994.

Kutluhan Erol, James A Hendler, and Dana S Nau. *Semantics for Hierarchical Task-Network Planning*. University of Maryland College Park, 1994.

Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD*, 1996.

Richard Evans and Edward Grefenstette. Learning Explanatory Rules From Noisy Data. *JAIR*, 61:1–64, 2018.

Kuan Fang, Yuke Zhu, Animesh Garg, Andrey Kurenkov, Viraj Mehta, Li Fei-Fei, and Silvio Savarese. Learning Task-Oriented Grasping for Tool Manipulation From Simulated Self-Supervision. *IJRR*, 39(2-3):202–216, 2020.

Kuan Fang, Fangchen Liu, Pieter Abbeel, and Sergey Levine. MOKA: Open-World Robotic Manipulation Through Mark-Based Visual Prompting. *RSS*, 2024.

Xiaolin Fang, Caelan Reed Garrett, Clemens Eppner, Tomás Lozano-Pérez, Leslie Pack Kaelbling, and Dieter Fox. DiMSam: Diffusion Models as Samplers for Task and Motion Planning Under Partial Observability. In *ICRA*, 2024.

Xiaolin Fang, Bo-Ruei Huang, Jiayuan Mao, Jasmine Shone, Joshua B Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Keypoint Abstraction Using Large Models for Object-Relative Imitation Learning. In *ICRA*, 2025.

Afsaneh Fazly, Afra Alishahi, and Suzanne Stevenson. A Probabilistic Computational Model of Cross-Situational Word Learning. *Cognitive Science*, 34(6):1017–1063, 2010.

Matthias Fey and Jan E. Lenssen. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Richard E Fikes and Nils J Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *AIJ*, 2(3-4):189–208, 1971.

Chelsea Finn and Sergey Levine. Deep Visual Foresight for Planning Robot Motion. In *ICRA*, 2017.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In *ICML*, 2017.

Jerry Fodor and Brian P McLaughlin. Connectionism and the Problem of Systematicity: Why Smolensky's Solution Doesn't Work. *Cognition*, 35(2):183–204, 1990.

Jerry A Fodor and Zenon W Pylyshyn. Connectionism and Cognitive Architecture: A Critical Analysis. *Cognition*, 28(1-2):3–71, 1988.

Maria Fox and Derek Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR*, 20:61–124, 2003.

Michael C Frank and Noah D Goodman. Predicting Pragmatic Reasoning in Language Games. *Science*, 336(6084):998–998, 2012.

Eugene C Freuder. A Sufficient Condition for Backtrack-Free Search. *JACM*, 29(1):24–32, 1982.

Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning Probabilistic Relational Models. In *IJCAI*, 1999.

Stephanie Fu, Mark Hamilton, Laura E. Brandt, Axel Feldmann, Zhoutong Zhang, and William T. Freeman. FeatUp: A Model-Agnostic Framework for Features at Any Resolution. In *ICLR*, 2024.

Chuang Gan, Yandong Li, Haoxiang Li, Chen Sun, and Boqing Gong. VQS: Linking Segmentations to Questions and Answers for Supervised Attention in Vqa and Question-Focused Semantic Segmentation. In *ICCV*, 2017.

Octavian-Eugen Ganea, Gary Bécigneul, and Thomas Hofmann. Hyperbolic Entailment Cones for Learning Hierarchical Embeddings. In *ICML*, 2018.

Siddha Ganju, Olga Russakovsky, and Abhinav Gupta. What's in a Question: Using Visual Questions as a Form of Supervision. In *CVPR*, 2017.

Wei Gao and Russ Tedrake. kPAM 2.0: Feedback Control for Category-Level Robotic Manipulation. *RA-L*, 6(2):2962–2969, 2021.

Marta Garnelo, Kai Arulkumaran, and Murray Shanahan. Towards Deep Symbolic Reinforcement Learning. *arXiv:1609.05518*, 2016.

Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Sampling-Based Methods for Factored Task and Motion Planning. *IJRR*, 37(13-14):1796–1825, 2018.

Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. PDDLStream: Integrating Symbolic Planners and Blackbox Samplers via Optimistic Adaptive Planning. In *ICAPS*, 2020.

Caelan Reed Garrett, Chris Paxton, Tomás Lozano-Pérez, Leslie Pack Kaelbling, and Dieter Fox. Online Replanning in Belief Space for Partially Observable Task and Motion Problems. In *ICRA*, 2020.

Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrated Task and Motion Planning. *Annual review of control, robotics, and autonomous systems*, 4(1):265–293, 2021.

Jon Gauthier, Roger Levy, and Joshua B Tenenbaum. Word Learning and the Acquisition of Syntactic–Semantic Overhypotheses. In *CogSci*, 2018.

Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional Sequence to Sequence Learning. In *ICML*, 2017.

Robert Geirhos, Patricia Rubisch, Claudio Michaelis, Matthias Bethge, Felix A. Wichmann, and Wieland Brendel. ImageNet-trained CNNs Are Biased Towards Texture; Increasing Shape Bias Improves Accuracy and Robustness. In *ICLR*, 2019.

Theophile Gervet, Zhou Xian, Nikolaos Gkanatsios, and Katerina Fragkiadaki. Act3D: 3D Feature Field Transformers for Multi-Task Robotic Manipulation. In *CORL*, 2023.

Spyros Gidaris and Nikos Komodakis. Dynamic Few-Shot Visual Learning Without Forgetting. In *CVPR*, 2018.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural Message Passing for Quantum Chemistry. In *ICML*, 2017.

Nikolaos Gkanatsios, Ayush Jain, Zhou Xian, Yunchu Zhang, Christopher Atkeson, and Katerina Fragkiadaki. Energy-Based Models as Zero-Shot Planners for Compositional Scene Rearrangement. In *RSS*, 2023.

Afonso Gonçalves, Giovanni Saponaro, Lorenzo Jamone, and Alexandre Bernardino. Learning Visual Affordances of Objects and Tools Through Autonomous Robot Exploration. In *ICARSC*, 2014.

Marco Gori, Gabriele Monfardini, and Franco Scarselli. A New Model for Learning in Graph Domains. In *IJCNN*, 2005.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. *arXiv:1410.5401*, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid Computing Using a Neural Network with Dynamic External Memory. *Nature*, 538(7626):471, 2016.

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging Pre-Trained Large Language Models to Construct and Utilize World Models for Model-Based Task Planning. In *NeurIPS*, 2023.

Carlos Guestrin, Daphne Koller, Chris Gearhart, and Neal Kanodia. Generalizing Plans to New Environments in Relational MDPs. In *IJCAI*, 2003.

Demi Guo, Yoon Kim, and Alexander Rush. Sequence-Level Mixed Sample Data Augmentation. In *EMNLP*, 2020.

Meng-Hao Guo, Jun-Xiong Cai, Zheng-Ning Liu, Tai-Jiang Mu, Ralph R Martin, and Shi-Min Hu. PCT: Point Cloud Transformer. *Computational Visual Media*, 7:187–199, 2021.

Abhinav Gupta and Larry S Davis. Objects in Action: An Approach for Combining Action Understanding and Object Perception. In *CVPR*, 2007.

Tanmay Gupta and Aniruddha Kembhavi. Visual Programming: Compositional Visual Reasoning Without Training. In *CVPR*, 2023.

Naresh Gupta and Dana S Nau. On the Complexity of Blocks-World Planning. *AIJ*, 56(2-3):223–254, 1992.

Jung-Su Ha, Danny Driess, and Marc Toussaint. Deep Visual Constraints: Neural Implicit Models For Manipulation Planning From Visual Input. *RA-L*, 2022.

Robert F Hadley. Systematicity in Connectionist Language Learning. *Mind & Language*, 9(3):273–287, 1994.

Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering Atari with Discrete World Models. In *ICLR*, 2021.

William L. Hamilton, Rex Ying, and Jure Leskovec. Representation Learning on Graphs: Methods and Applications. *arxiv:1709.05584*, 2017.

Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In Yoshua Bengio and Yann LeCun, editors, *ICLR*, 2016.

Chi Han, Jiayuan Mao, Chuang Gan, Joshua B. Tenenbaum, and Jiajun Wu. Visual Concept Metaconcept Learning. In *NeurIPS*, 2019.

Muzhi Han, Yifeng Zhu, Song-Chun Zhu, Ying Nian Wu, and Yuke Zhu. InterPreT: Interactive Predicate Learning From Language Feedback for Generalizable Task Planning. In *RSS*, 2024.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with Language Model Is Planning with World Model. In *EMNLP*, 2023.

Juris Hartmanis and Richard E Stearns. On the Computational Complexity of Algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

Kris Hauser and Jean-Claude Latombe. Multi-Modal Motion Planning in Non-Expansive Spaces. *IJRR*, 29(7):897–915, 2010.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2015.

Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask R-Cnn. In *ICCV*, 2017.

Malte Helmert. The Fast Downward Planning System. *JAIR*, 26:191–246, 2006.

Irina Higgins, Nicolas Sonnerat, Loic Matthey, Arka Pal, Christopher P Burgess, Matthew Botvinick, Demis Hassabis, and Alexander Lerchner. SCAN: Learning Abstract Hierarchical Compositional Visual Concepts. In *ICLR*, 2018.

Nathan Hilliard, Lawrence Phillips, Scott Howland, Artëm Yankov, Courtney D. Corley, and Nathan O. Hodas. Few-Shot Learning with Metric-Agnostic Conditional Embeddings. *arxiv:1802.04376*, 2018.

Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *Signal Processing*, 29(6):82–97, 2012.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Comput.*, 9(8):1735–1780, 1997.

Jörg Hoffmann and Bernhard Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *JAIR*, 14:253–302, 2001.

Yining Hong, Yilun Du, Chunru Lin, Joshua B. Tenenbaum, and Chuang Gan. 3D Concept Grounding on Neural Fields. In *NeurIPS*, 2022.

Alfred Horn. On Sentences Which Are True of Direct Unions of Algebras. *Journal of Symbolic Logic*, 16(1):14–21, 1951.

Patrik O. Hoyer. Non-Negative Matrix Factorization with Sparseness Constraints. *JMLR*, 5:1457–1469, 2004.

Joy Hsu, Jiayuan Mao, Joshua B. Tenenbaum, and Jiajun Wu. What's Left? Concept Grounding with Logic-Enhanced Foundation Models. In *NeurIPS*, 2023.

Joy Hsu, Jiayuan Mao, and Jiajun Wu. NS3D: Neuro-Symbolic Grounding of 3d Objects and Relations. In *CVPR*, pages 2614–2623, 2023.

Yuxiao Hu and Giuseppe De Giacomo. Generalized Planning: Synthesizing Plans That Work for Multiple Environments. In *IJCAI*, 2011.

Ronghang Hu, Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Kate Saenko. Learning to Reason: End-to-End Module Networks for Visual Question Answering. In *CVPR*, 2017.

Ronghang Hu, Jacob Andreas, Trevor Darrell, and Kate Saenko. Explainable Neural Computation via Stack Neural Module Networks. In *ECCV*, 2018.

Yingdong Hu, Fanqi Lin, Tong Zhang, Li Yi, and Yang Gao. Look Before You Leap: Unveiling the Power of GPT-4V in Robotic Vision-Language Planning. *arXiv:2311.17842*, 2023.

Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. Densely Connected Convolutional Networks. In *CVPR*, 2017.

De-An Huang, Suraj Nair, Danfei Xu, Yuke Zhu, Animesh Garg, Li Fei-Fei, Silvio Savarese, and Juan Carlos Niebles. Neural Task Graphs: Generalizing to Unseen Tasks From a Single Video Demonstration. In *CVPR*, 2019.

De-An Huang, Danfei Xu, Yuke Zhu, Animesh Garg, Silvio Savarese, Fei-Fei Li, and Juan Carlos Niebles. Continuous Relaxation of Symbolic Planner for One-Shot Imitation Learning. In *IROS*, 2019.

Jiani Huang, Ziyang Li, Binghong Chen, Karan Samel, Mayur Naik, Le Song, and Xujie Si. Scallop: From Probabilistic Deductive Databases to Scalable Differentiable Reasoning. In *NeurIPS*, 2021.

Shijia Huang, Yilun Chen, Jiaya Jia, and Liwei Wang. Multi-View Transformer for 3D Visual Grounding. In *CVPR*, 2022.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents. In *ICML*. PMLR, 2022.

Eric Huang, Xianyi Cheng, Yuemin Mao, Arnav Gupta, and Matthew T Mason. Autogenerated Manipulation Primitives. *IJRR*, 2023.

Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. VoxPoser: Composable 3D Value Maps for Robotic Manipulation with Language Models. In *CORL*, 2023.

Wenlong Huang, Chen Wang, Yunzhu Li, Ruohan Zhang, and Li Fei-Fei. ReKep: Spatio-Temporal Reasoning of Relational Keypoint Constraints for Robotic Manipulation. In *CORL*, 2024.

Yixuan Huang, Nichols Crawford Taylor, Adam Conkey, Weiyu Liu, and Tucker Hermans. Latent Space Planning for Multi-Object Manipulation with Environment-Aware Relational Classifiers. *T-RO*, 2024.

Drew A Hudson and Christopher D Manning. Compositional Attention Networks for Machine Reasoning. In *ICLR*, 2018.

Drew A Hudson and Christopher D Manning. GQA: A New Dataset for Real-World Visual Reasoning and Compositional Question Answering. In *CVPR*, 2019.

Drew A Hudson and Christopher D Manning. Learning by Abstraction: The Neural State Machine. In *NeurIPS*, 2019.

Niall P. Hurley and Scott T. Rickard. Comparing Measures of Sparsity. *IEEE Transactions on Information Theory*, 55(10):4723–4741, 2009.

Mostafa S. Ibrahim, Srikanth Muralidharan, Zhiwei Deng, Arash Vahdat, and Greg Mori. A Hierarchical Deep Temporal Model for Group Activity Recognition. In *CVPR*, 2016.

Stephen S Intille and Aaron F Bobick. A Framework for Recognizing Multi-Agent Action From Visual Evidence. In *AAAI*, 1999.

Stephen S Intille and Aaron F Bobick. Recognizing Planned, Multiperson Action. *CVIU*, 81(3):414–445, 2001.

Allan Jabri, Armand Joulin, and Laurens van der Maaten. Revisiting Visual Question Answering Baselines. In *ECCV*, 2016.

Ayush Jain, Nikolaos Gkanatsios, Ishita Mediratta, and Katerina Fragkiadaki. Bottom Up Top Down Detection Transformers for Language Grounding in Images and Point Clouds. In *ECCV*, 2022.

Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J. Davison. RLBench: The Robot Learning Benchmark & Learning Environment. *RA-L*, 2020.

Michael Janner, Yilun Du, Joshua B. Tenenbaum, and Sergey Levine. Planning with Diffusion for Flexible Behavior Synthesis. In *ICML*, 2022.

Peter A Jansen and Scott Watter. Strong Systematicity Through Sensorimotor Conceptual Grounding: An Unsupervised, Developmental Approach to Connectionist Sentence Processing. *Connect. Sci.*, 24(1):25–55, 2012.

C Greg Jensen, W Edward Red, and J Pi. Tool Selection for Five-Axis Curvature Matched Machining. *Computer-Aided Design*, 34(3):251–266, 2002.

Nikolay Jetchev, Tobias Lang, and Marc Toussaint. Learning Grounded Relational Symbols From Continuous Data for Abstract Reasoning. In *ICRA Workshop*, 2013.

Xuerong Ji and Jing Xiao. Planning Motions Compliant to Complex Contact States. *IJRR*, 20(6):446–465, 2001.

Zhengyao Jiang and Shan Luo. Neural Logic Reinforcement Learning. In *ICML*, 2019.

Sergio Jiménez, Javier Segovia-Aguas, and Anders Jonsson. A Review of Generalized Planning. *The Knowledge Engineering Review*, 34, 2019.

Lifeng Jin and William Schuler. Grounded PCFG Induction with Images. In *AACL*, 2020.

Justin Johnson, Ranjay Krishna, Michael Stark, Li-Jia Li, David Shamma, Michael Bernstein, and Li Fei-Fei. Image Retrieval Using Scene Graphs. In *CVPR*, 2015.

Justin Johnson, Andrej Karpathy, and Li Fei-Fei. DenseCap: Fully Convolutional Localization Networks for Dense Captioning. In *CVPR*, 2016.

Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. CLEVR: A Diagnostic Dataset for Compositional Language and Elementary Visual Reasoning. In *CVPR*, 2017.

Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Judy Hoffman, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Inferring and Executing Programs for Visual Reasoning. In *ICCV*, 2017.

Yuanchen Ju, Kaizhe Hu, Guowei Zhang, Gu Zhang, Mingrun Jiang, and Huazhe Xu. Robo-ABC: Affordance Generalization Beyond Categories via Semantic Correspondence for Robot Manipulation. In *ECCV*, 2024.

Leslie Pack Kaelbling and Tomás Lozano-Pérez. Hierarchical Task and Motion Planning in the Now. In *ICRA*, pages 1470–1477, 2011.

Leslie Pack Kaelbling. Hierarchical Learning in Stochastic Domains: Preliminary Results. In *ICML*, 1993.

Leslie Pack Kaelbling. Learning to Achieve Goals. In *IJCAI*, 1993.

Łukasz Kaiser and Ilya Sutskever. Neural GPUs Learn Algorithms. In *ICLR*, 2016.

Aishwarya Kamath, Mannat Singh, Yann LeCun, Gabriel Synnaeve, Ishan Misra, and Nicolas Carion. MDETR – Modulated Detection for End-to-End Multi-Modal Understanding. In *ICCV*, 2021.

Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Kaya Stechly, Mudit Verma, Siddhant Bhambri, Lucas Saldyt, and Anil Murthy. LLMs Can't Plan, but Can Help Planning in LLM-Modulo Frameworks. In *ICML*, 2024.

Michael Kampffmeyer, Yinbo Chen, Xiaodan Liang, Hao Wang, Yujia Zhang, and Eric P Xing. Rethinking Knowledge Graph Propagation for Zero-Shot Learning. In *CVPR*, 2019.

Ken Kansky, Tom Silver, David A Mély, Mohamed Eldawy, Miguel Lázaro-Gredilla, Xinghua Lou, Nimrod Dorfman, Szymon Sidor, Scott Phoenix, and Dileep George. Schema Networks: Zero-Shot Transfer with a Generative Causal Model of Intuitive Physics. In *ICML*, 2017.

Richard M Karp. *Reducibility Among Combinatorial Problems*. Springer, 1972.

Tadao Kasami. An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages. *Coordinated Science Laboratory Report no. R-257*, 1966.

Tsung-Wei Ke, Nikolaos Gkanatsios, and Katerina Fragkiadaki. 3D Diffuser Actor: Policy Diffusion with 3D Scene Representations. In *CORL*, 2024.

Kristian Kersting, Luc De Raedt, and Stefan Kramer. Interpreting Bayesian Logic Programs. In *AAAI Workshop on Learning Statistical Models from Relational Data*, 2000.

Yoon Kim, Chris Dyer, and Alexander M Rush. Compound Probabilistic Context-Free Grammars for Grammar Induction. In *ACL*, 2019.

Angelika Kimmig, Stephen Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. A Short Introduction to Probabilistic Soft Logic. In *NeurIPS Workshop on Probabilistic Programming: Foundations and Applications*, 2012.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *ICLR*, 2015.

Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*, 2017.

Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alex Berg, Wan-Yen Lo, Piotr Dollar, and Ross Girshick. Segment Anything. In *ICCV*, 2023.

Ryan Kiros, Ruslan Salakhutdinov, and Richard S Zemel. Unifying Visual-Semantic Embeddings with Multimodal Neural Language Models. *arXiv:1411.2539*, 2014.

Po-Chen Ko, Jiayuan Mao, Yilun Du, Shao-Hua Sun, and Joshua B. Tenenbaum. Learning to Act from Actionless Video through Dense Correspondences. In *ICLR*, 2024.

George Konidaris, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. From Skills to Symbols: Learning Symbolic Representations for Abstract High-Level Planning. *JAIR*, 61:215–289, 2018.

Richard E Korf. Planning as Search: A Quantitative Approach. *AIJ*, 33(1):65–88, 1987.

Janne H Korhonen and Jukka Suomela. Towards a Complexity Theory for the Congested Clique. In *SPAA*, 2018.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification with Deep Convolutional Neural Networks. In *NeurIPS*, 2012.

Tejas D Kulkarni, William F Whitney, Pushmeet Kohli, and Josh Tenenbaum. Deep Convolutional Inverse Graphics Network. In *NeurIPS*, 2015.

Zihang Lai, Senthil Purushwalkam, and Abhinav Gupta. The Functional Correspondence Problem. In *ICCV*, 2021.

Brenden Lake and Marco Baroni. Generalization Without Systematicity: On the Compositional Skills of Sequence-to-Sequence Recurrentlake Networks. In *ICML*, 2018.

Luis C. Lamb, Rafael V. Borges, and A. d'Avila Garcez. A Connectionist Cognitive Model for Temporal Synchronisation and Learning. In *AAAI*, 2007.

Steven M LaValle and James J Kuffner Jr. Randomized Kinodynamic Planning. *IJRR*, 20(5):378–400, 2001.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proc. IEEE*, 86(11):2278–2324, 1998.

Gilwoo Lee, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Hierarchical Planning for Multi-Contact Non-Prehensile Manipulation. In *IROS*, 2015.

AA Leman and B Weisfeiler. A Reduction of a Graph to a Canonical Form and an Algebra Arising During This Reduction. *Nauchno-Technicheskaya Informatsiya*, 2(9):12–16, 1968.

Hector J Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *The Journal of Logic Programming*, 31(1-3):59–83, 1997.

Hector J Levesque. Planning with Loops. In *IJCAI*, pages 509–515, 2005.

Beth Levin. *English Verb Classes and Alternations*, volume 1. University of Chicago Press, 1993.

Lihong Li, Thomas J Walsh, and Michael L Littman. Towards a Unified Theory of State Abstraction for MDPs. *ISAIM*, 4(5):9, 2006.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated Graph Sequence Neural Networks. In *ICLR*, 2016.

Qing Li, Siyuan Huang, Yining Hong, Yixin Chen, Ying Nian Wu, and Song-Chun Zhu. Closed Loop Neural-Symbolic Learning via Integrating Neural Perception, Grammar Parsing, and Symbolic Reasoning. In *ICML*, 2020.

Qing Li, Siyuan Huang, Yining Hong, and Song-Chun Zhu. A Competence-Aware Curriculum for Visual Concepts Learning via Question Answering. In *ECCV*, 2020.

Richard Li, Allan Jabri, Trevor Darrell, and Pulkit Agrawal. Towards Practical Multi-Object Manipulation Using Relational Reinforcement Learning. In *ICRA*, 2020.

Qi Li, Kaichun Mo, Yanchao Yang, Hang Zhao, and Leonidas Guibas. IFR-Explore: Learning Inter-Object Functional Relationships in 3D Indoor Scenes. In *ICLR*, 2022.

Zhaoyi Li, Kelin Yu, Shuo Cheng, and Danfei Xu. League++: Empowering Continual Robot Learning Through Guided Skill Acquisition with Large Language Models. In *ICLR Workshop on Large Language Model (LLM) Agents*, 2024.

Li, Xiang and Vilnis, Luke and Zhang, Dongxu and Boratko, Michael and McCallum, Andrew. Smoothing the Geometry of Probabilistic Box Embeddings. In *ICLR*, 2019.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as Policies: Language Model Programs for Embodied Control. In *ICRA*, 2023.

Vladimir Lifschitz. On the Semantics of STRIPS. In *Reasoning about Actions and Plans*. Morgan Kaufmann, 1987.

Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B Tenenbaum, and Stephen H Muggleton. Bias Reformulation for One-Shot Function Induction. In *ECAI*, 2014.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common Objects in Context. In *ECCV*, 2014.

Xingyu Lin, Carl Qi, Yunchu Zhang, Zhiao Huang, Katerina Fragkiadaki, Yunzhu Li, Chuang Gan, and David Held. Planning with Spatial and Temporal Abstraction From Point Clouds for Deformable Object Manipulation. In *CoRL*, 2022.

Kevin Lin, Christopher Agia, Toki Migimatsu, Marco Pavone, and Jeannette Bohg. Text2Motion: From Natural Language Instructions to Feasible Plans. *Autonomous Robots*, 47(8):1345–1365, 2023.

Nir Lipovetzky and Hector Geffner. Width and Serialization of Classical Planning Problems. In *ECAI*. IOS Press, 2012.

Marco Lippi and Paolo Frasconi. Prediction of Protein β-Residue Contacts by Markov Logic Networks with Grounding-Specific Weights. *Bioinformatics*, 25(18):2326–2333, 2009.

Xiao Liu, Jia-min Liu, An-xi Cao, and Zhuang-le Yao. HAPE3D—A New Constructive Algorithm for the 3D Irregular Packing Problem. *Frontiers of Information Technology & Electronic Engineering*, 16(5):380–390, 2015.

Runtao Liu, Chenxi Liu, Yutong Bai, and Alan L Yuille. CLEVR-Ref+: Diagnosing Visual Reasoning with Referring Expressions. In *CVPR*, pages 4185–4194, 2019.

Runtao Liu, Chenxi Liu, Yutong Bai, and Alan L. Yuille. CLEVR-Ref+: Diagnosing Visual Reasoning With Referring Expressions. In *CVPR*, 2019.

Guiliang Liu, Ashutosh Adhikari, Amir-massoud Farahmand, and Pascal Poupart. Learning Object-Oriented Dynamics for Planning From Text. In *ICLR*, 2021.

Nan Liu, Shuang Li, Yilun Du, Joshua B. Tenenbaum, and Antonio Torralba. Learning to Compose Visual Relations. In *NeurIPS*, 2021.

Nan Liu, Shuang Li, Yilun Du, Antonio Torralba, and Joshua B Tenenbaum. Compositional Visual Generation with Composable Diffusion Models. In *ECCV*, 2022.

Weiyu Liu, Chris Paxton, Tucker Hermans, and Dieter Fox. StructFormer: Learning Spatial Structure for Language-Guided Semantic Rearrangement of Novel Objects. In *ICRA*, 2022.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. LLM+P: Empowering Large Language Models with Optimal Planning Proficiency. *arXiv:2304.11477*, 2023.

Weiyu Liu, Yilun Du, Tucker Hermans, Sonia Chernova, and Chris Paxton. StructDiffusion: Language-Guided Creation of Physically-Valid Structures Using Unseen Objects. In *RSS*, 2023.

Weiyu Liu, Jiayuan Mao, Joy Hsu, Tucker Hermans, Animesh Garg, and Jiajun Wu. Composable Part-Based Manipulation. In *CoRL*, 2023.

Shilong Liu, Zhaoyang Zeng, Tianhe Ren, Feng Li, Hao Zhang, Jie Yang, Chunyuan Li, Jianwei Yang, Hang Su, Jun Zhu, et al. Grounding DINO: Marrying DINO with Grounded Pre-Training for Open-Set Object Detection. In *ECCV*, 2024.

Weiyu Liu, Neil Nie, Ruohan Zhang, Jiayuan Mao, and Jiajun Wu. BLADE: Learning Compositional Behaviors From Demonstration and Language. In *CoRL*, 2024.

Yuyao Liu, Jiayuan Mao, Joshua Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. One-Shot Manipulation Strategy Learning by Making Contact Analogies. In *ICRA*, 2025.

Zhezheng Luo Luo, Jiayuan Mao, Joshua B. Tenenbaum, and Leslie Pack Kaelbling. On the Expressiveness and Learning of Relational Neural Networks on Hypergraphs. In *Learning on Graphs*, 2021.

Zhezheng Luo, Jiayuan Mao, Joshua B. Tenenbaum, and Leslie Pack Kaelbling. On the Expressiveness and Generalization of Hypergraph Neural Networks. In *LoG*, 2022.

Zhezheng Luo, Jiayuan Mao, Joshua B Tenenbaum, and Leslie Pack Kaelbling. On the Expressiveness and Generalization of Hypergraph Neural Networks. In *Learning on Graphs*, 2022.

Jianlan Luo, Charles Xu, Xinyang Geng, Gilbert Feng, Kuan Fang, Liam Tan, Stefan Schaal, and Sergey Levine. Multi-Stage Cable Routing Through Hierarchical Imitation Learning. *T-RO*, 2024.

Corey Lynch and Pierre Sermanet. Language Conditioned Imitation Learning Over Unstructured Data. In *RSS*, 2021.

Corey Lynch, Mohi Khansari, Ted Xiao, Vikash Kumar, Jonathan Tompson, Sergey Levine, and Pierre Sermanet. Learning Latent Plans From Play. In *CoRL*, 2020.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-Level Reward Design via Coding Large Language Models. In *ICLR*, 2024.

David Maier and David S. Warren. *Computing with Logic: Logic Programming with Prolog*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1988.

Jean Maillard and Stephen Clark. Latent Tree Learning with Differentiable Parsers: Shift-Reduce Parsing and Chart Parsing. In *ACL Workshop*, 2018.

M Malinowski and M Fritz. A Multi-World Approach to Question Answering about Real-World Scenes Based on Uncertain Input. In *NeurIPS*, 2014.

Ajay Mandlekar, Caelan Garrett, Danfei Xu, and Dieter Fox. Human-in-the-Loop Task and Motion Planning for Imitation Learning. In *CoRL*, 2023.

Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. DeepProbLog: Neural Probabilistic Logic Programming. In *NeurIPS*, 2018.

Lucas Manuelli, Wei Gao, Peter R. Florence, and Russ Tedrake. kPAM: KeyPoint Affordances for Category-Level Robotic Manipulation. In *ISRR*, 2019.

Junhua Mao, Jiajing Xu, Kevin Jing, and Alan L Yuille. Training and Evaluating Multimodal Word Embeddings with Large-Scale Web Annotated Images. In *NeurIPS*, 2016.

Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision. In *ICLR*, 2019.

Jiayuan Mao, Zhezheng Luo, Chuang Gan, Joshua B Tenenbaum, Jiajun Wu, Leslie Pack Kaelbling, and Tomer D Ullman. Temporal and Object Quantification Networks. In *IJCAI*, 2021.

Jiayuan Mao, Haoyue Shi, Jiajun Wu, Roger P. Levy, and Joshua B. Tenenbaum. Grammar-Based Grounded Lexicon Learning. In *NeurIPS*, 2021.

Jiayuan Mao, Tomás Lozano-Pérez, Joshua B. Tenenbaum, and Leslie Kaelbling. PDSketch: Integrated Domain Programming, Learning, and Planning. In *NeurIPS*, 2022.

Jiayuan Mao, Tomás Lozano-Pérez, Joshua B. Tenenbaum, and Leslie Kaelbling. What Planning Problems Can a Relational Neural Network Solve? In *NeurIPS*, 2023.

Jiayuan Mao, Tomás Lozano-Pérez, Joshua B. Tenenbaum, and Leslie Pack Kaelbling. Learning Reusable Manipulation Strategies. In *CoRL*, 2023.

Jiayuan Mao, Tomás Lozano-Pérez, Joshua B. Tenenbaum, and Leslie Pack Kaelbling. What Planning Problem Can A Relational Neural Network Solve. In *NeurIPS*, 2023.

Jiayuan Mao, Joshua B. Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Hybrid Declarative-Imperative Representations for Hybrid Discrete-Continuous Decision-Making. In *WAFR*, 2024.

Eric Margolis and Stephen Laurence. *Concepts: Core Readings*. The MIT Press, 1999.

David Mascharka, Philip Tran, Ryan Soklaski, and Arjun Majumdar. Transparency by Design: Closing the Gap Between Performance and Interpretability in Visual Reasoning. In *CVPR*, 2018.

Matthew T Mason. *Mechanics of Robotic Manipulation*. MIT press, 2001.

Angelos Mavrogiannis, Christoforos Mavrogiannis, and Yiannis Aloimonos. Cook2LTL: Translating Cooking Recipes to LTL Formulae Using Large Language Models. In *ICRA*, 2024.

Oier Mees, Lukás Hermann, Erick Rosete-Beas, and Wolfram Burgard. CALVIN: A Benchmark for Language-Conditioned Policy Learning for Long-Horizon Robot Manipulation Tasks. *RA-L*, 7:7327–7334, 2021.

Oier Mees, Lukas Hermann, and Wolfram Burgard. What Matters in Language Conditioned Robotic Imitation Learning Over Unstructured Data. *RA-L*, 2022.

Oier Mees, Jessica Borja-Diaz, and Wolfram Burgard. Grounding Language with Visual Affordances Over Unstructured Data. In *ICRA*, 2023.

Lingjie Mei, Jiayuan Mao, Ziqi Wang, Chuang Gan, and Joshua B. Tenenbaum. FALCON: Fast Visual Concept Learning by Integrating Images, Linguistic Descriptions, and Conceptual Relations. In *ICLR*, 2022.

Christian Meilicke, Manuel Fink, Yanjie Wang, Daniel Ruffinelli, Rainer Gemulla, and Heiner Stuckenschmidt. Fine-Grained Evaluation of Rule- And Embedding-Based Systems for Knowledge Graph Completion. In *ISWC*, 2018.

Jorge Mendez-Mendez, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Embodied Lifelong Learning for Task and Motion Planning. In *CoRL*. PMLR, 2023.

Toki Migimatsu and Jeannette Bohg. Grounding Predicates Through Actions. In *ICRA*, 2022.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-Level Control Through Deep Reinforcement Learning. *Nature*, 518(7540):529–533, 02 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *ICML*, 2016.

Kaichun Mo, Leonidas J Guibas, Mustafa Mukadam, Abhinav Gupta, and Shubham Tulsiani. Where2Act: From Pixels to Actions for Articulated 3D Objects. In *CVPR*, 2021.

Igor Mordatch, Emanuel Todorov, and Zoran Popović. Discovery of Complex Behaviors Through Contact-Invariant Optimization. *ToG*, 31(4):1–8, 2012.

Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks. In *AAAI*, 2019.

Arsalan Mousavian, Clemens Eppner, and Dieter Fox. 6-DoF GraspNet: Variational Grasp Generation for Object Manipulation. In *ICCV*, 2019.

Stephen Muggleton. Inductive Logic Programming. *New Gener. Comput.*, 8(4):295–318, 1991.

Stephen Muggleton. Stochastic Logic Programs. *Advances in Inductive Logic Programming*, 32:254–264, 1996.

Tsendsuren Munkhdalai and Hong Yu. Meta Networks. In *ICML*, 2017.

Austin Myers, Ching L Teo, Cornelia Fermüller, and Yiannis Aloimonos. Affordance Detection of Tool Parts From Geometric Features. In *ICRA*, 2015.

Cota Nabeshima, Yasuo Kuniyoshi, and Max Lungarella. Towards a Model for Tool-Body Assimilation and Adaptive Tool-Use. In *ICDL*, 2007.

Suraj Nair and Chelsea Finn. Hierarchical Foresight: Self-Supervised Learning of Long-Horizon Tasks via Visual Subgoal Generation. In *ICLR*, 2020.

Soroush Nasiriany, Vitchyr Pong, Steven Lin, and Sergey Levine. Planning with Goal-Conditioned Policies. In *NeurIPS*, 2019.

Dana S Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN Planning System. *JAIR*, 20:379–404, 2003.

Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural Programmer: Inducing Latent Programs with Gradient Descent. In *ICLR*, 2016.

Daniel Neider and Ivan Gavran. Learning Linear Temporal Properties. In *FMCAD*, 2018.

Allen Newell, Herbert Alexander Simon, et al. *Human Problem Solving*, volume 104:9. Prentice-hall Englewood Cliffs, NJ, 1972.

Van-Duc Nguyen. *The Synthesis of Force-Closure Grasps*. PhD thesis, Massachusetts Institute of Technology, 1985.

Alex Nichol and Prafulla Dhariwal. Improved Denoising Diffusion Probabilistic Models, 2021.

Alex Nichol, Joshua Achiam, and John Schulman. On First-Order Meta-Learning Algorithms. *arxiv:1803.02999*, 2018.

Weili Nie, Arash Vahdat, and Anima Anandkumar. Controllable and Compositional Generation with Latent-Space Energy-Based Models. In *NeurIPS*, 2021.

Scott Niekum, Sarah Osentoski, George Konidaris, and Andrew G Barto. Learning and Generalization of Complex Tasks From Unstructured Demonstrations. In *IROS*, 2012.

Nils J Nilsson. *Principles of Artificial Intelligence*. Springer Science & Business Media, 1982.

Kolby Nottingham, Prithviraj Ammanabrolu, Alane Suhr, Yejin Choi, Hannaneh Hajishirzi, Sameer Singh, and Roy Fox. Do Embodied Agents Dream of Pixelated Sheep: Embodied Decision Making Using Language Guided World Modelling. In *ICML*, 2023.

OpenAI. Hello GPT-4o, 2024. Accessed: 2024-09-15.

Maxime Oquab, Timothée Darcet, Théo Moutakanni, Huy Vo, Marc Szafraniec, Vasil Khalidov, Pierre Fernandez, Daniel Haziza, Francisco Massa, Alaaeldin El-Nouby, et al. DINOv2: Learning Robust Visual Features Without Supervision. *TMLR*, 2024.

Tao Pang, HJ Terry Suh, Lujie Yang, and Russ Tedrake. Global Planning for Contact-Rich Manipulation via Local Smoothing of Quasi-Dynamic Contact Models. *T-RO*, 2023.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-Symbolic Program Synthesis. In *ICLR*, 2017.

Hanna M Pasula, Luke S Zettlemoyer, and Leslie Pack Kaelbling. Learning Symbolic Models of Stochastic Domains. *JAIR*, 29:309–352, 2007.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*, 2019.

Chris Paxton, Chris Xie, Tucker Hermans, and Dieter Fox. Predicting Stable Configurations for Semantic Placement of Novel Objects. In *CoRL*, 2022.

Leo de Penning, A. d'Avila Garcez, Luís C. Lamb, and John-Jules C. Meyer. A Neural-Symbolic Cognitive Agent for Online Learning and Reasoning. In *IJCAI*, 2011.

Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global Vectors for Word Representation. In *EMNLP*, 2014.

Ethan Perez, Florian Strub, Harm De Vries, Vincent Dumoulin, and Aaron Courville. FiLM: Visual Reasoning with a General Conditioning Layer. In *AAAI*, 2018.

Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. Learning with Algorithmic Supervision via Continuous Relaxations. In *NeurIPS*, 2021.

Sören Pirk, Karol Hausman, Alexander Toshev, and Mohi Khansari. Modeling Long-Horizon Tasks as Sequential Interaction Landscapes. In *CoRL*, 2020.

Amir Pnueli. The Temporal Logic of Programs. In *FOCS*, 1977.

Carl Pollard and Ivan Sag. *Head-Driven Phrase Structure Grammar*. Chicago: University of Chicago Press and Stanford: CSLI Publications., 1994.

Vitchyr Pong, Shixiang Gu, Murtaza Dalal, and Sergey Levine. Temporal Difference Models: Model-Free Deep RL for Model-Based Control. In *ICLR*, 2018.

Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space, 2017.

Mengshi Qi, Jie Qin, Annan Li, Yunhong Wang, Jiebo Luo, and Luc Van Gool. stagNet: An Attentive Semantic RNN for Group Activity Recognition. In *ECCV*, 2018.

Zengyi Qin, Kuan Fang, Yuke Zhu, Li Fei-Fei, and Silvio Savarese. KETO: Learning Keypoint Representations for Tool Manipulation. In *ICRA*, 2020.

J. Ross Quinlan. Learning Logical Definitions From Relations. *MLJ*, 5(3):239–266, 1990.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning Transferable Visual Models From Natural Language Supervision. In *ICML*, 2021.

Yongming Rao, Wenliang Zhao, Guangyi Chen, Yansong Tang, Zheng Zhu, Guan Huang, Jie Zhou, and Jiwen Lu. DenseCLIP: Language-Guided Dense Prediction with Context-Aware Prompting. In *CVPR*, 2022.

Sachin Ravi and Hugo Larochelle. Optimization as a Model for Few-Shot Learning. In *ICLR*, 2017.

Scott Reed and Nando De Freitas. Neural Programmer-Interpreters. In *ICLR*, 2016.

Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In *NeurIPS*, 2015.

Hongyu Ren, Weihua Hu, and Jure Leskovec. Query2box: Reasoning Over Knowledge Graphs in Vector Space Using Box Embeddings. In *ICLR*, 2020.

Matthew Richardson and Pedro Domingos. Markov Logic Networks. *MLJ*, 62(1-2):107–136, 2006.

Ryan Riegel, Alexander Gray, Francois Luus, Naweed Khan, Ndivhuwo Makondo, Ismail Yunus Akhalwaya, Haifeng Qian, Ronald Fagin, Francisco Barahona, Udit Sharma, Shajith Ikbal, Hima Karanam, Sumit Neelam, Ankita Likhyani, and Santosh Srivastava. Logical Neural Networks. In *NeurIPS*, 2020.

Gareth O Roberts and Richard L Tweedie. Exponential Convergence of Langevin Distributions and Their Discrete Approximations. *Bernoulli*, pages 341–363, 1996.

Tim Rocktäschel and Sebastian Riedel. End-to-End Differentiable Proving. In *NeurIPS*, 2017.

Diego Rodriguez and Sven Behnke. Transferring Category-Based Functional Grasping Skills by Latent Space Non-Rigid Registration. *RA-L*, 3(3):2662–2669, 2018.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *MICCAI*, 2015.

Candace Ross, Andrei Barbu, Yevgeni Berzak, Battushig Myanganbayar, and Boris Katz. Grounding Language Acquisition by Training Semantic Parsers Using Captioned Videos. In *EMNLP*, 2018.

Radu Bogdan Rusu, Nico Blodow, and Michael Beetz. Fast Point Feature Histograms (FPFH) for 3D Registration. In *ICRA*, 2009.

Earl D Sacerdoti. Planning in a Hierarchy of Abstraction Spaces. *AIJ*, 5(2):115–135, 1974.

Earl D Sacerdoti. The Nonlinear Nature of Plans, 1975.

Ali Sadeghian, Mohammadreza Armandpour, Patrick Ding, and Daisy Zhe Wang. DRUM: End-to-End Differentiable Rule Mining on Knowledge Graphs. In *NeurIPS*, 2019.

Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-Learning with Memory-Augmented Neural Networks. In *ICML*, 2016.

Adam Santoro, David Raposo, David GT Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Timothy Lillicrap. A Simple Neural Network Module for Relational Reasoning. In *NeurIPS*, 2017.

Raeid Saqur and Karthik Narasimhan. Multimodal Graph Networks for Compositional Generalization in Visual Question Answering. In *NeurIPS*, 2020.

Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal Value Function Approximators. In *ICML*, 2015.

Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling Relational Data with Graph Convolutional Networks. In *Extended Semantic Web Conference (ESWC)*, 2018.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *Nature*, 588(7839):604–609, 2020.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347*, 2017.

Sebastian Schuster, Ranjay Krishna, Angel Chang, Li Fei-Fei, and Christopher D Manning. Generating Semantically Precise Scene Graphs From Textual Descriptions for Improved Image Retrieval. In *EMNLP Workshop*, 2015.

Alberto Segre and Gerald DeJong. Explanation-Based Manipulator Learning: Acquisition of Planning Ability Through Observation. In *ICRA*, 1985.

Naman Shah, Jayesh Nagpal, Pulkit Verma, and Siddharth Srivastava. From Reals to Logic and Back: Inventing Symbolic Vocabularies, Actions and Models for Planning From Raw Data. *arXiv:2402.11871*, 2024.

William Shen, Ge Yang, Alan Yu, Jansen Wong, Leslie Pack Kaelbling, and Phillip Isola. Distilled Feature Fields Enable Few-Shot Language-Guided Manipulation. In *CORL*, 2023.

Haoyue Shi, Jiayuan Mao, Tete Xiao, Yuning Jiang, and Jian Sun. Learning Visually-Grounded Semantics From Contrastive Adversarial Samples . In *COLING*, 2018.

Haoyue Shi, Jiayuan Mao, Kevin Gimpel, and Karen Livescu. Visually Grounded Neural Syntax Acquisition. In *ACL*, 2019.

Lei Shi, Yifan Zhang, Jian Cheng, and Hanqing Lu. Two-Stream Adaptive Graph Convolutional Networks for Skeleton-Based Action Recognition. In *CVPR*, 2019.

Tianze Shi, Ozan Irsoy, Igor Malioutov, and Lillian Lee. Learning Syntax From Naturally-Occurring Bracketings. In *NAACL-HLT*, 2021.

Haochen Shi, Huazhe Xu, Samuel Clarke, Yunzhu Li, and Jiajun Wu. RoboCook: Long-Horizon Elasto-Plastic Object Manipulation with Diverse Tools. In *CoRL*, 2023.

Lucy Xiaoyang Shi, Zheyuan Hu, Tony Z Zhao, Archit Sharma, Karl Pertsch, Jianlan Luo, Sergey Levine, and Chelsea Finn. Yell at Your Robot: Improving on-the-Fly From Language Corrections. In *RSS*, 2024.

Mohit Shridhar, Lucas Manuelli, and Dieter Fox. CLIPort: What and Where Pathways for Robotic Manipulation. In *CORL*, 2022.

Mohit Shridhar, Lucas Manuelli, and Dieter Fox. Perceiver-Actor: A Multi-Task Transformer for Robotic Manipulation. In *CORL*, 2022.

N Siddharth, T. B. Paige, J.W. Meent, A. Desmaison, N. Goodman, P. Kohli, F. Wood, and P. Torr. Learning Disentangled Representations with Semi-Supervised Deep Generative Models. In *NeurIPS*, 2017.

Maximilian Sieb, Zhou Xian, Audrey Huang, Oliver Kroemer, and Katerina Fragkiadaki. Graph-Structured Visual Imitation. In *CoRL*, 2020.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the Game of Go Without Human Knowledge. *Nature*, 550(7676):354, 2017.

Tom Silver, Rohan Chitnis, Joshua B. Tenenbaum, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Learning Symbolic Operators for Task and Motion Planning. In *IROS*, 2021.

Tom Silver, Varun Hariprasad, Reece S Shuttleworth, Nishanth Kumar, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. PDDL Planning with Pretrained Large Language Models. In *NeurIPS foundation models for decision making workshop*, 2022.

Tom Silver, Rohan Chitnis, Nishanth Kumar, Willie McClinton, Tomás Lozano-Pérez, Leslie Kaelbling, and Joshua B Tenenbaum. Predicate Invention for Bilevel Planning. In *AAAI*, 2023.

Tom Silver, Soham Dan, Kavitha Srinivas, Joshua B Tenenbaum, Leslie Kaelbling, and Michael Katz. Generalized Planning in PDDL Domains with Pretrained Large Language Models. In *AAAI*, 2024.

Anthony Simeonov, Yilun Du, Beomjoon Kim, Francois R Hogan, Joshua Tenenbaum, Pulkit Agrawal, and Alberto Rodriguez. A Long Horizon Planning Framework for Manipulating Rigid Pointcloud Objects. In *CoRL*, 2020.

Anthony Simeonov, Yilun Du, Andrea Tagliasacchi, Joshua B Tenenbaum, Alberto Rodriguez, Pulkit Agrawal, and Vincent Sitzmann. Neural Descriptor Fields: SE(3)-Equivariant Object Representations for Manipulation. In *ICRA*, 2022.

Anthony Simeonov, Yilun Du, Lin Yen-Chen, , Alberto Rodriguez, Leslie P. Kaelbling, Tomás Lozano-Pérez, and Pulkit Agrawal. SE(3)-Equivariant Relational Rearrangement with Neural Descriptor Fields. In *CoRL*, 2022.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. ProgPrompt: Generating Situated Robot Task Plans Using Large Language Models. In *ICRA*, 2023.

Marta Skreta, Zihan Zhou, Jia Lin Yuan, Kourosh Darvish, Alán Aspuru-Guzik, and Animesh Garg. RePLan: Robotic Replanning with Perception and Language Models. *arXiv:2401.04157*, 2024.

Jean-Pierre Sleiman, Jan Carius, Ruben Grandia, Martin Wermelinger, and Marco Hutter. Contact-Implicit Trajectory Optimization for Dynamic Object Manipulation. In *IROS*, 2019.

Pavel Smirnov, Frank Joublin, Antonello Ceravola, and Michael Gienger. Generating Consistent PDDL Domains with Large Language Models. *arXiv:2404.07751*, 2024.

Jake Snell, Kevin Swersky, and Richard S Zemel. Prototypical Networks for Few-Shot Learning. In *NeurIPS*, 2017.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *EMNLP*, 2013.

Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.

Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. A New Representation and Associated Algorithms for Generalized Planning. *AIJ*, 175(2):615–647, 2011.

Siddharth Srivastava, Eugene Fang, Lorenzo Riano, Rohan Chitnis, Stuart Russell, and Pieter Abbeel. Combined Task and Motion Planning Through an Extensible Planner-Independent Interface Layer. In *ICRA*, 2014.

Mark Steedman. *Surface Structure and Interpretation*. MIT press, 1996.

Mark Steedman. *The Syntactic Process*. Cambridge, MA: MIT Press, 2000.

Mike Stilman and James J Kuffner. Navigation Among Movable Obstacles: Real-Time Reasoning in Complex Environments. *International Journal of Humanoid Robotics*, 2(04):479–503, 2005.

Alexander Stoytchev. Behavior-Grounded Representation of Tool Affordances. In *ICRA*, 2005.

Jürgen Stränger and Bernhard Hommel. The Perception of Action and Movement. In *Handbook of perception and action*, volume 1, pages 397–451. Elsevier, 1996.

Joseph Suarez, Justin Johnson, and Fei-Fei Li. DDRprog: A CLEVR Differentiable Dynamic Reasoning Programmer. *arxiv:1803.11361*, 2018.

Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. End-to-End Memory Networks. In *NeurIPS*, 2015.

Brian L. Sullivan, Christopher L. Wood, Marshall J. Iliff, Rick E. Bonney, Daniel Fink, and Steve Kelling. eBird: A Citizen-Based Bird Observation Network in the Biological Sciences. *Biological Conservation*, 142(10):2282 – 2292, 2009.

Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. Neural Program Synthesis From Diverse Demonstration Videos. In *ICML*, 2018.

Shao-Hua Sun, Te-Lin Wu, and Joseph J Lim. Program Guided Agent. In *ICLR*, 2019.

Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. RotatE: Knowledge Graph Embedding by Relational Rotation in Complex Space. In *ICLR*, 2019.

Flood Sung, Yongxin Yang, Li Zhang, Tao Xiang, Philip H.S. Torr, and Timothy M. Hospedales. Learning to Compare: Relation Network for Few-Shot Learning. In *CVPR*, 2018.

Dídac Surís, Sachit Menon, and Carl Vondrick. ViperGPT: Visual Inference via Python Execution for Reasoning. In *ICCV*, 2023.

Gerald J Sussman. A Computational Model of Skill Acquisition. *AI Technical Reports*, 1973.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to Sequence Learning with Neural Networks. In *NeurIPS*, 2014.

Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*, volume 1. MIT press Cambridge, 1998.

Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *NeurIPS*, 2000.

Swingley, Daniel. Fast Mapping and Slow Mapping in Children's Word Learning. *Language Learning and Development*, 6:179–183, 07 2010.

Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *ACL*, 2015.

Kevin Tang, Li Fei-Fei, and Daphne Koller. Learning Latent Temporal Structure for Complex Event Detection. In *CVPR*, 2012.

Luming Tang, Menglin Jia, Qianqian Wang, Cheng Perng Phoo, and Bharath Hariharan. Emergent Correspondence From Image Diffusion. In *NeurIPS*, 2023.

Marjorie Taylor and Susan A Gelman. Adjectives and Nouns: Children's Strategies for Learning New Words. *Child Development*, pages 411–419, 1988.

Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew Walter, Ashis Banerjee, Seth Teller, and Nicholas Roy. Understanding Natural Language Commands for Robotic Navigation and Mobile Manipulation. In *AAAI*, 2011.

Komal Teru, Etienne Denis, and Will Hamilton. Inductive Relation Prediction by Subgraph Reasoning. In *ICML*, 2020.

Skye Thompson, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Shape-Based Transfer of Generic Skills. In *ICRA*, 2021.

Yonglong Tian, Yue Wang, Dilip Krishnan, Tenenbaum Joshua B., and Phillip Isola. Rethinking Few-Shot Image Classification: A Good Embedding Is All You Need? In *ECCV*, 2020.

Marc Toussaint. Logic-Geometric Programming: An Optimization-Based Approach to Combined Task and Motion Planning. In *IJCAI*, 2015.

Kristina Toutanova, Danqi Chen, Patrick Pantel, Hoifung Poon, Pallavi Choudhury, and Michael Gamon. Representing Text for Joint Embedding of Text and Knowledge Bases. In Lluís Màrquez, Chris Callison-Burch, Jian Su, Daniele Pighin, and Yuval Marton, editors, *EMNLP*, 2015.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. LLaMA: Open and Efficient Foundation Language Models. *arXiv:2302.13971*, 2023.

Son D Tran and Larry S Davis. Event Modeling and Recognition Using Markov Logic Networks. In *ECCV*, 2008.

Jeffrey C Trinkle and Jerry J Hunter. A Framework for Planning Dexterous Manipulation. In *ICRA*, 1991.

Théo Trouillon, Christopher R. Dance, Éric Gaussier, Johannes Welbl, Sebastian Riedel, and Guillaume Bouchard. Knowledge Graph Completion via Complex Tensor Factorization. *JMLR*, 2017.

Maria Tsimpoukelli, Jacob L Menick, Serkan Cabi, SM Eslami, Oriol Vinyals, and Felix Hill. Multimodal Few-Shot Learning with Frozen Language Models. In *NeurIPS*, 2021.

Dylan Turpin, Liquan Wang, Stavros Tsogkas, Sven J. Dickinson, and Animesh Garg. GIFT: Generalizable Interaction-Aware Functional Tool Affordances Without Labels. In *RSS*, 2021.

Julen Urain, Anqi Li, Puze Liu, Carlo D'Eramo, and Jan Peters. Composable Energy Policies for Reactive Motion Generation and Reinforcement Learning. In *RSS*, 2021.

Julen Urain, Niklas Funk, Georgia Chalvatzaki, and Jan Peters. SE(3)-DiffusionFields: Learning Cost Functions for Joint Grasp and Motion Optimization Through Diffusion. In *ICRA*, 2023.

Nikolaus Vahrenkamp, Leonard Westkamp, Natsuki Yamanobe, Eren E Aksoy, and Tamim Asfour. Part-Based Grasp Planning for Familiar Objects. In *Humanoids*, 2016.

Robin R Vallacher and Daniel M Wegner. What Do People Think They're Doing? Action Identification and Human Behavior. *Psychological Review*, 94(1):3, 1987.

Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. On the Planning Abilities of Large Language Models-a Critical Investigation. In *NeurIPS*, 2023.

Aaron Van Den Oord, Oriol Vinyals, et al. Neural Discrete Representation Learning. In *NeurIPS*, 2017.

Martijn Van Otterlo. *The Logic of Adaptive Behavior*. IOS Press, 2009.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *NeurIPS*, 2017.

Rishi Veerapaneni, John D Co-Reyes, Michael Chang, Michael Janner, Chelsea Finn, Jiajun Wu, Joshua Tenenbaum, and Sergey Levine. Entity Abstraction in Visual Model-Based Reinforcement Learning. In *CORL*, 2020.

William Vega-Brown and Nicholas Roy. Task and Motion Planning Is PSPACE-Complete. In *AAAI*, 2020.

Ivan Vendrov, Ryan Kiros, Sanja Fidler, and Raquel Urtasun. Order-Embeddings of Images and Language. In *ICLR*, 2016.

Luke Vilnis and Andrew McCallum. Word Representations via Gaussian Embedding. In *ICLR*, 2015.

Luke Vilnis, Xiang Li, Shikhar Murty, and Andrew McCallum. Probabilistic Embedding of Knowledge Graphs with Box Lattice Measures. In *ACL*, 2018.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer Networks. In *NeurIPS*, 2015.

Oriol Vinyals, Charles Blundell, Tim Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. Matching Networks for One Shot Learning. In *NeurIPS*, 2016.

Catherine Wah, Steve Branson, Peter Welinder, Pietro Perona, and Serge Belongie. The Caltech-Ucsd Birds-200-2011 Dataset. Technical report, California Institute of Technology, 2011.

Naoki Wake, Atsushi Kanehira, Kazuhiro Sasabuchi, Jun Takamatsu, and Katsushi Ikeuchi. ChatGPT Empowered Long-Step Robot Control in Various Environments: A Case Application. *IEEE Access*, 2023.

Thomas J Walsh. *Efficient Learning of Relational Models for Sequential Decision Making*. PhD thesis, Rutgers The State University of New Jersey-New Brunswick, 2010.

Weikang Wan, Yifeng Zhu, Rutav Shah, and Yuke Zhu. LOTUS: Continual Imitation Learning for Robot Manipulation Through Unsupervised Skill Discovery. In *ICRA*, 2024.

Xiaolong Wang and Abhinav Gupta. Videos as Space-Time Region Graphs. In *ECCV*, 2018.

Fan Wang and Kris Hauser. Stable Bin Packing of Non-Convex 3D Objects with a Robot Manipulator. In *ICRA*, 2019.

Tingwu Wang, Renjie Liao, Jimmy Ba, and Sanja Fidler. NerveNet: Learning Structured Policy with Graph Neural Networks. In *ICLR*, 2018.

Xiaolong Wang, Ross Girshick, Abhinav Gupta, and Kaiming He. Non-Local Neural Networks. In *CVPR*, 2018.

Xiaolong Wang, Yufei Ye, and Abhinav Gupta. Zero-Shot Recognition via Semantic Embeddings and Knowledge Graphs. In *CVPR*, 2018.

Xinyi Wang, Hieu Pham, Zihang Dai, and Graham Neubig. SwitchOut: An Efficient Data Augmentation Algorithm for Neural Machine Translation. In *EMNLP*, 2018.

Chen Wang, Rui Wang, Ajay Mandlekar, Li Fei-Fei, Silvio Savarese, and Danfei Xu. Generalization Through Hand-Eye Coordination: An Action Space for Learning Spatially-Invariant Visuomotor Control. In *IROS*, 2021.

Peng Wang, An Yang, Rui Men, Junyang Lin, Shuai Bai, Zhikang Li, Jianxin Ma, Chang Zhou, Jingren Zhou, and Hongxia Yang. OFA: Unifying Architectures, Tasks, and Modalities Through a Simple Sequence-to-Sequence Learning Framework. In *ICML*, 2022.

Chen Wang, Linxi Fan, Jiankai Sun, Ruohan Zhang, Li Fei-Fei, Danfei Xu, Yuke Zhu, and Anima Anandkumar. MimicPlay: Long-Horizon Imitation Learning by Watching Human Play. In *CoRL*, 2023.

Renhao Wang, Jiayuan Mao, Joy Hsu, Hang Zhao, Jiajun Wu, and Yang Gao. Programmatically Grounded, Compositionally Generalizable Robotic Manipulation. In *ICLR*, 2023.

Huaxiaoyue Wang, Kushal Kedia, Juntao Ren, Rahma Abdullah, Atiksh Bhardwaj, Angela Chao, Kelly Y Chen, Nathaniel Chin, Prithwish Dan, Xinyi Fan, et al. MOSAIC: A Modular System for Assistive and Interactive Cooking. *arXiv:2402.18796*, 2024.

Yanwei Wang, Tsun-Hsuan Wang, Jiayuan Mao, Michael Hagenow, and Julie Shah. Grounding Language Plans in Demonstrations Through Counterfactual Perturbations. In *ICLR*, 2024.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*, 2022.

Bowen Wen, Wenzhao Lian, Kostas E. Bekris, and Stefan Schaal. You Only Demonstrate Once: Category-Level Manipulation From Single Visual Demonstration. In *RSS*, 2022.

Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks. *arXiv:1502.05698*, 2015.

Ronald J Williams and Jing Peng. Function Optimization Using Connectionist Reinforcement Learning Algorithms. *Connect. Sci.*, 3(3):241–268, 1991.

Ronald J Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *MLJ*, 8(3-4):229–256, 1992.

Lionel Wong, Jiayuan Mao, Pratyusha Sharma, Zachary S. Siegel, Jiahai Feng, Noa Korneev, Joshua B. Tenenbaum, and Jacob Andreas. Learning Adaptive Planning Representations with Natural Language Guidance. In *ICLR*, 2024.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation. *arXiv:1609.08144*, 2016.

Jiajun Wu, Joshua B Tenenbaum, and Pushmeet Kohli. Neural Scene De-Rendering. In *CVPR*, 2017.

Tailin Wu, Megan Tjandrasuwita, Zhengxuan Wu, Xuelin Yang, Kevin Liu, Rok Sosic, and Jure Leskovec. ZeroC: A Neuro-Symbolic Model for Zero-Shot Concept Recognition and Acquisition at Inference Time. In *NeurIPS*, 2022.

Zhenyu Wu, Ziwei Wang, Xiuwei Xu, Jiwen Lu, and Haibin Yan. Embodied Task Planning with Large Language Models. *arXiv:2307.01848*, 2023.

Victoria Xia, Zi Wang, and Leslie Pack Kaelbling. Learning Sparse Relational Transition Models. In *ICLR*, 2019.

Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, Li Yi, Angel X. Chang, Leonidas J. Guibas, and Hao Su. SAPIEN: A SimulAted Part-Based Interactive ENvironment. In *CVPR*, 2020.

Jiannan Xiang, Tianhua Tao, Yi Gu, Tianmin Shu, Zirui Wang, Zichao Yang, and Zhiting Hu. Language Models Meet World Models: Embodied Experiences Enhance Language Models. In *NeurIPS*, 2024.

Jing Xiao and Xuerong Ji. Automatic Generation of High-Level Contact State Space. *IJRR*, 20(7):584–606, 2001.

Guangxuan Xiao, Leslie Pack Kaelbling, Jiajun Wu, and Jiayuan Mao. Sparse and Local Hypergraph Reasoning Networks. In *LoG*, 2022.

Yaqi Xie, Chen Yu, Tongyao Zhu, Jinbin Bai, Ze Gong, and Harold Soh. Translating Natural Language to Planning Goals with Large-Language Models. *arXiv:2302.05128*, 2023.

Wenhan Xiong, Thien Hoang, and William Yang Wang. DeepPath: A Reinforcement Learning Method for Knowledge Graph Reasoning. In Martha Palmer, Rebecca Hwa, and Sebastian Riedel, editors, *EMNLP*, pages 564–573. Association for Computational Linguistics, 2017.

Huijuan Xu and Kate Saenko. Ask, Attend and Answer: Exploring Question-Guided Spatial Attention for Visual Question Answering. In *ECCV*, 2016.

Danfei Xu, Roberto Martín-Martín, De-An Huang, Yuke Zhu, Silvio Savarese, and Li Fei-Fei. Regression Planning Networks. In *NeurIPS*, 2019.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful Are Graph Neural Networks? In *ICLR*, 2019.

Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What Can Neural Networks Reason About? *ICLR*, 2020.

Keyulu Xu, Mozhi Zhang, Jingling Li, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. How Neural Networks Extrapolate: From Feedforward to Graph Neural Networks. In *ICLR*, 2021.

Ruinian Xu, Fu-Jen Chu, Chao Tang, Weiyu Liu, and Patricio A Vela. An Affordance Keypoint Detection Network for Robot Manipulation. *RA-L*, 6(2):2870–2877, 2021.

Sijie Yan, Yuanjun Xiong, and Dahua Lin. Spatial Temporal Graph Convolutional Networks for Skeleton-Based Action Recognition. In *AAAI*, 2018.

Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding Entities and Relations for Learning and Inference in Knowledge Bases. In *ICLR*, 2015.

Jimei Yang, Scott E Reed, Ming-Hsuan Yang, and Honglak Lee. Weakly-Supervised Disentangling with Recurrent Transformations for 3d View Synthesis. In *NeurIPS*, 2015.

Zichao Yang, Xiaodong He, Jianfeng Gao, Li Deng, and Alex Smola. Stacked Attention Networks for Image Question Answering. In *CVPR*, 2016.

Fan Yang, Zhilin Yang, and William W. Cohen. Differentiable Learning of Logical Rules for Knowledge Base Reasoning. In *NeurIPS*, 2017.

Fan Yang, Zhilin Yang, and William W Cohen. Differentiable Learning of Logical Rules for Knowledge Base Reasoning. In *NeurIPS*, 2017.

Huanrui Yang, Wei Wen, and Hai Li. DeepHoyer: Learning Sparser Neural Network with Differentiable Scale-Invariant Sparsity Measures. In *ICLR*, 2020.

Zhengyuan Yang, Songyang Zhang, Liwei Wang, and Jiebo Luo. SAT: 2D Semantics Assisted Training for 3D Visual Grounding. In *ICCV*, 2021.

Zhutian Yang, Caelan Reed Garrett, Tomás Lozano-Pérez, Leslie Kaelbling, and Dieter Fox. Sequence-Based Plan Feasibility Prediction for Efficient Task and Motion Planning. In *RSS*, 2023.

Zhutian Yang, Jiayuan Mao, Yilun Du, Jiajun Wu, Joshua B. Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Compositional Diffusion-Based Continuous Constraint Solvers. In *CoRL*, 2023.

Masahito Yashima, Yoshikazu Shiina, and Hideya Yamaguchi. Randomized Manipulation Planning for a Multi-Fingered Hand by Switching Contact Modes. In *ICRA*, 2003.

Morris Yau, Eykin Akyurek, Jiayuan Mao, Joshua B. Tenenbaum, Stefanie Jegelka, and Jacob Andreas. Learning Linear Attention in Polynomial Time. In *ICML*, 2025.

Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Joshua B Tenenbaum. Neural-Symbolic VQA: Disentangling Reasoning From Vision and Language Understanding. In *NeurIPS*, 2018.

Daniel H Younger. Recognition and Parsing of Context-Free Languages in Time N3. *Information and control*, 10(2):189–208, 1967.

Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-World: A Benchmark and Evaluation for Multi-Task and Meta Reinforcement Learning. In *CORL*, 2019.

Wentao Yuan, Chris Paxton, Karthik Desingh, and Dieter Fox. SORNet: Spatial Object-Centric Representations for Sequential Manipulation. In *CoRL*, 2021.

Jeffrey M Zacks, Barbara Tversky, and Gowri Iyer. Perceiving, Remembering, and Communicating Structure in Events. *Journal of Experimental Psychology: General*, 130(1):29, 2001.

Jeffrey M Zacks, Nicole K Speer, Khena M Swallow, Todd S Braver, and Jeremy R Reynolds. Event Perception: A Mind-Brain Perspective. *Psychological Bulletin*, 133(2):273, 2007.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D Goodman, and Nick Haber. Parsel: A (De-)Compositional Framework for Algorithmic Reasoning with Language Models. In *NeurIPS*, 2023.

Wenyuan Zeng, Yankai Lin, Zhiyuan Liu, and Maosong Sun. Incorporating Relation Paths in Neural Relation Extraction. In *EMNLP*, 2017.

Andy Zeng, Pete Florence, Jonathan Tompson, Stefan Welker, Jonathan Chien, Maria Attarian, Travis Armstrong, Ivan Krasin, Dan Duong, Vikas Sindhwani, et al. Transporter Networks: Rearranging the Visual World for Robotic Manipulation. In *CoRL*, 2020.

Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *ICLR*, 2020.

Luke Zettlemoyer and Michael Collins. Learning Context-Dependent Mappings From Sentences to Logical Form. In *ACL*, 2009.

Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. Mixup: Beyond Empirical Risk Minimization. In *ICLR*, 2017.

Amy Zhang, Rowan McAllister, Roberto Calandra, Yarin Gal, and Sergey Levine. Learning Invariant Representations for Reinforcement Learning Without Reconstruction. In *ICLR*, 2021.

Junyi Zhang, Charles Herrmann, Junhwa Hur, Luisa Polania Cabrera, Varun Jampani, Deqing Sun, and Ming-Hsuan Yang. A Tale of Two Features: Stable Diffusion Complements DINO for Zero-Shot Semantic Correspondence. In *NeurIPS*, 2023.

Youcai Zhang, Xinyu Huang, Jinyu Ma, Zhaoyang Li, Zhaochuan Luo, Yanchun Xie, Yuzhuo Qin, Tong Luo, Yaqian Li, Shilong Liu, et al. Recognize Anything: A Strong Image Tagging Model. In *CVPR*, 2024.

Zichen Zhang, Yunshuang Li, Osbert Bastani, Abhishek Gupta, Dinesh Jayaraman, Yecheng Jason Ma, and Luca Weihs. Universal Visual Decomposer: Long-Horizon Manipulation Made Easy. In *ICRA*, 2024.

Yanpeng Zhao and Ivan Titov. Visually Grounded Compound PCFGs. In *EMNLP*, 2020.

Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A Modern Library for 3D Data Processing. *arXiv:1801.09847*, 2018.

Yuke Zhu, Alireza Fathi, and Li Fei-Fei. Reasoning about Object Affordances in a Knowledge Base Representation. In *ECCV*, 2014.

Xiaodan Zhu, Parinaz Sobihani, and Hongyu Guo. Long Short-Term Memory Over Recursive Structures. In *ICML*, 2015.

Yixin Zhu, Yibiao Zhao, and Song-Chun Zhu. Understanding Tools: Task-Oriented Object Modeling, Learning, and Recognition. In *CVPR*, 2015.

Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning Books and Movies: Towards Story-Like Visual Explanations by Watching Movies and Reading Books. In *CVPR*, 2015.

Guangxiang Zhu, Zhiao Huang, and Chongjie Zhang. Object-Oriented Dynamics Predictor. In *NeurIPS*, 2018.

Yifeng Zhu, Jonathan Tremblay, Stan Birchfield, and Yuke Zhu. Hierarchical Planning for Long-Horizon Manipulation with Geometric and Symbolic Scene Graphs. In *ICRA*, 2020.

Yifeng Zhu, Peter Stone, and Yuke Zhu. Bottom-Up Skill Discovery From Unsegmented Demonstrations for Long-Horizon Robot Manipulation. *RA-L*, 7(2):4126–4133, 2022.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, et al. Ghost in the Minecraft: Generally Capable Agents for Open-World Enviroments via Large Language Models with Text-Based Knowledge and Memory. *arXiv:2305.17144*, 2023.

Yifeng Zhu, Abhishek Joshi, Peter Stone, and Yuke Zhu. VIOLA: Imitation Learning for Vision-Based Manipulation with Object Proposal Priors. In *CORL*, 2023.

Appendix A

Appendix for Neural Logic Machines

This chapter is organized as follows. First, we provide more details for our training method and introduce the curriculum learning used for reinforcement learning tasks in Appendix A.1. Second, in Appendix A.2, we provide more implementation details and hyper-parameters of each task in Section 2.3. Next, we provide deferred discussion of NLM extensions in Appendix A.3. Besides, we give a proof of how NLMs could realize the the forward chaining of a set of logic rules defined in Horn cluases. See Appendix A.4 for details. In Appendix A.5, We also provide a list of sample rules for the blocks world problem in order to exhibit the complexity of describing a strategies or policies. Finally, we also provide a minimal implementation of NLM in TensorFlow for reference at the end of the chapter (Appendix A.6).

A.1 Training Method and Curriculum Learning

In this section, we provide hyper-parameter details of our training method and introduce the exam-guided curriculum learning used for reinforcement learning tasks. We also provide details of the data generation method.

A.1.1 Training method

We optimize both NLM and MemNN with Adam (Kingma and Ba [2015]) and use a learning rate of $\alpha = 0.005$.

For all supervised learning tasks (i.e. family tree and general graph tasks), we use Softmax-Cross-Entropy as loss function and a training batch size of 4.

For reinforcement learning tasks (i.e. the blocks world, sorting and shortest path tasks), we use REINFORCE algorithm (Sutton and Barto [1998]) for optimization. Each training batch is composed of a single episode of play. Similar to A3C (Mnih *et al.* [2016]), we add

policy entropy term in the objective function (proposed by Williams and Peng [1991]) to help exploration. The update function for parameters θ of policy π is

$$\Delta\theta = \alpha[v_t \nabla_\theta \log \pi(a_t|s_t;\theta) + \beta \nabla_\theta H(\pi(s_t;\theta))],$$

where H is the entropy function, s_t and a_t are the state and action at time t, v_t is the discounted reward starting from time t. The hyper-parameter β is set according to different environments and learning stages depending on the demand of exploration.

In all environments, the agent receives a reward of value 1.0 when it completes the task within a limited number of steps (which is related to the number of objects). To encourage the agent to use as few moves as possible, we give a reward of -0.01 for each move. The reward discount factor γ is 0.99 for all tasks.

Table A.1: **Hyper-parameters for reinforcement learning tasks.** The meaning of the hyperparameters could be found in Section A.1.1 and Section A.1.2. For the `Path` environment, the step limit is set to the actual distance between the starting point and the targeting point, to encourage the agents to find the shortest path.

Task	Range	Step Limit	β_{init}	Ω	Epochs	Train Epoch Episodes	Evaluation Episodes
`Sorting`	$m \in [4, 10]$	$2m$	0.01	0.5	5	200	200
`Path`	$m \in [3, 12]$	opt	0.1	0.5	40	600	3000
`Blocks World`	$m \in [2, 12]$	$4m$	0.2	0.6	50	1000	3000

A.1.2 Curriculum learning guided by exams and fails

Algorithm 10 Curriculum learning guided by exams and fails

```
function TRAIN(model M, lessons 𝓛)
  for ℓ ∈ 𝓛 do
    for i = 0, 1, ··· ℓ.max_epochs do
      accuracy, pos, neg ← evaluate(M, ℓ)        ▷ Take the exam and collect samples.
      if accuracy > ℓ.threshold then
        break                                    ▷ Enter the next lesson if pass the exam.
      for j = 0, 1, ··· K do
        data ~ balanced sampling from pos and neg
        Optimize M with data
```

Inspired by the education system of humans, we employ an exam-guided curriculum learning [Bengio *et al.*, 2009] approach for training Neural Logic Machines. We heuristically

label each training instances with its complexity. Training instances are grouped by their complexity (as *lessons*). For example, in the game of BlocksWorld, we consider the number of blocks in a game instance as its complexity. During the training, we present the training instances to the model from lessons with increasing difficulty. We periodically test models' performance (as *exams*) on novel instances of the same complexity as the ones in its current lesson. The well-performed model (whose accuracy reaches a certain threshold) will *pass* the exam and advance to a harder lesson (of more complex training instances). The exam-guided curriculum learning exploits the previously gained knowledge to ease the learning of more complex instances. Moreover, the performance on the final exam reaches above a threshold indicates the *graduation* of models.

In our experiments, each lesson contains training instances of the same number of objects. For example, the first lesson in the blocks world contains all possible instances consisting of 2 blocks (in each world). The instances of the second lesson contain 3 blocks in each world. And in the last lesson (totally 11 lessons) there are 12 blocks in each world. We report the range of the curriculum in Table A.1 for three RL tasks.

Another essential ingredient for the efficient training of NLMs is to record models' failure cases. Specifically, we keep track of two sets of training instances: positive and negative (meaning the agent achieves the task or not). For each presented instance of the exam, it is recollected into positive or negative sets depending on whether the agent achieves the task or not. All training samples are sampled from the positive set with probability Ω and from the negative set with probability $1 - \Omega$. This balanced sampling strategy prevents models from getting stuck at sub-optimal solutions. Algorithm 10 illustrates the pseudo-code of the curriculum learning guided by exams and fails.

The evaluation process ("exam") randomly samples examples from 3 recent lessons. The agent goes through these examples and gets the success rate (the ratio of achieving the task) as its performance, which is used to decide whether the agent passes the exam by comparing to a lesson-depend threshold. As we want a perfect model, the threshold for passing the last lesson (the "final exam") is 100%. We linearly decrease the threshold by 0.5% for each former lessons, to prevent over-fitting(e.g., the threshold of the first lesson in the blocks world is 95%). After the "exam", the examples are collected into positive and negative pools according to the outcome (success or not). During the training, we use balanced sampling for choosing training instances from positive and negative pools with probability Ω from positive. The hyper-parameters Ω, the number of epochs, the number of episodes in each training epoch and the number of episodes in one evaluation are shown in Table A.1 for three RL tasks.

A.2 Implementation details and hyper-parameters

This section provides more implementation details for the model and experiments, and summarizes the hyper-parameters used in experiments for our NLM and the baseline algorithm MemNN.

A.2.1 Residual connection.

Analog to the residual link in [He *et al.*, 2015; Huang *et al.*, 2017], we add residual connections to our model. Specifically, for each layer illustrated in Figure 2-2, the base predicates (inputs) are concatenated to the conclusive predicates (outputs) group-wisely. That is, input unary predicates are concatenated to the deduced unary predicates while input binary predicates are concatenated to the conclusive binary predicates.

A.2.2 Hyper-parameters for NLM

Table A.2 shows hyper-parameters used by NLM for different tasks. For all MLPs inside NLMs, we use no hidden layer, and the hidden dimension (i.e., the number of intermediate predicates) of each layer is set to 8 across all our experiments. In supervised learning tasks, a model is called "graduated" if its training loss is below a threshold depending on the task (usually 1e-6). In reinforcement learning tasks, an agent is called "graduated" if it can pass the final exam, i.e., get 100% success rate on the evaluation process of the last lesson.

We note that in the randomly generated cases, the number of maternal great uncle (`IsMGUncle`) relation is relatively small. This makes the learning of this relation hard and results in a graduation ratio of only 20%. If we increase the maximum number of people in training examples to 30, the graduation ratio will grow to 50%.

A.2.3 Hyper-parameters for MemNN

We set the number of iters/episodes used for baseline algorithms to be same as NLM. For the memory networks, each pre-condition in the memory is embedded into a key space and a value space. The dimensions of the spaces are 16 and 32 respectively. The hidden size of the LSTM in MemNN is 64. The number of queries is set to be 4 across all tasks (except that the `Sorting` task uses 1 query only). Empirically, we search for the optimal hyper-parameters but find that they have little effect on the performance.

Table A.2: **Hyper-parameters for Neural Logic Machines.** The definition of depth and breadth are illustrated in Figure 2-2. "Res." refers to the use of residual links. "Grad." refers to the ratio of successful graduation in 10 runs with different random seeds, which partially indicates the difficulty of the task. "Num. Examples/Episodes" means the maximum number of examples/episodes used to train the model in supervised learning and reinforcement learning cases.

	Tasks	Depth	Breath	Res.	Grad.	Num. Examples/Episodes
Family Tree	`HasFather`	4	3	×	100%	50,000 examples
	`HasSister`	4	3	×	100%	50,000 examples
	`IsGrandparent`	4	3	×	100%	100,000 examples
	`IsUncle`	4	3	×	90%	100,000 examples
	`IsMGUncle`	4	3	×	20%	200,000 examples
General Graph	`AdjacentToRed`	4	3	×	90%	100,000 examples
	`4-Connectivity`	4	3	×	100%	50,000 examples
	`6-Connectivity`	8	3	✓	60%	50,000 examples
	`1-OutDegree`	4	3	×	100%	50,000 examples
	`2-OutDegree`	5	4	✓	100%	100,000 examples
General Algorithm	`Sorting`	3	2	✓	100%	1,000 episodes
	`Path`	5	3	✓	60%	24,000 episodes
`BlocksWorld`		7	2	✓	40%	50,000 episodes

A.2.4 Data generation

We use random generation to generate training and testing data. more details and specific parameters used to generate the data could be found in our open source code.

In family tree tasks, we mimic the process of families growing using a timeline. For each newly created person, we randomly sample the gender and parents (could be none, indicating not included in the family tree) of the person. We also maintain lists of singles of each gender, and randomly pick two from each list to be married (each time when a person was created). We randomly permute the order of people.

In general graph tasks (include `Path`), We adopt the generation method from Graves *et al.* [2016], which samples m nodes on a unit square, and the out-degree k_i of each node is sampled. Then each node connects to k_i nearest nodes on the unit square. In undirected graph cases, all generated edges are regarded as undirected edges.

In `Sorting`, we randomly generate permutations to be sorted in ascending order.

In `Blocks World`, We maintain a list of placeable objects (the ground included). Each

newly created block places on one randomly selected placeable object. Then we randomly shuffle the *id* of the blocks.

A.2.5 Blocks World

In the blocks world environment, to better aid the reinforcement learning process, we train the agent on an auxiliary task, which is to predict the validity or effect of the actions. This task is trained by supervised learning using cross-entropy loss. The overall loss is a summation of cross-entropy loss (with a weight of 0.1) and the REINFORCE loss.

We did not choose the `Move` to be taken directly based on the relational predicates at the last layer of NLM. Instead, we manually concatenate the object representation from the current and the target configuration, which share the same object ID. Then for each pair of objects, their relational representation is constructed by the concatenation of their own object representation. An extra fully-connected layer is applied to the relational representation, followed by a `Softmax` layer over all pairs of objects. We choose an action based on the Softmax score.

A.2.6 Accuracy discussion

We cannot directly prove the accuracy of NLM by looking at the induced rules as in traditional ILP systems. Alternatively, we take an empirical way to estimate its accuracy by sampling testing examples. Throughout the experiments section, all accuracy statistics are reported in 1000 random generated data.

To show the confidence of this result, we test a specific trained model of `Blocks World` task with 100,000 samples. We get *no* fail cases in the testing. According to the multiplicative form of Chernoff Bound [*], We are 99.7% confident that the accuracy is at least 99.98%.

A.3 Neural Logic Machines (NLM) extensions

Reasoning over noisy input: integration with neural perception. Recall that NLM is fully differentiable. Besides taking logic pre-conditions (binary values) as input, the input properties or relations can be derived from other neural architectures (e.g., CNNs). As a preliminary example, we replace the input properties of nodes with images from the MNIST dataset. A convolutional neural network (CNN) is applied to the input, extracting multiple features for future reasoning. CNN and NLM can be optimized jointly. This enables reasoning over noisy input.

[*] https://en.wikipedia.org/wiki/Chernoff_bound#Multiplicative_form_(relative_error)

We modify the `AdjacentToRed` task in general graph reasoning to `AdjacentToNumber0`. In detail, each node has a visual input from the MNIST dataset indicating its number. We say `AdjacentToNumber0`(x) if and only if a node x is adjacent to another node with number 0. We use LeNet LeCun *et al.* [1998] to extract visual features for recognizing the number of each node. The output of LeNet for each node is a vector of length 10, with sigmoid activation.

We follow the train-test split from the original MNIST dataset. The joint model is trained on 100,000 training examples ($m = 10$) and gets 99.4% accuracy on 1000 testing examples ($m = 50$). Note that the LeNet modules are optimized jointly with the reasoning about `AdjacentToNumber0`.

A.4 Realization of Horn Clause

In this section, we show that *NLM can realize a partial set of Horn clauses [Horn, 1951] in first-order logic (FOL), up to the limit of the NLM's depth and breadth.* In NLMs, we consider only finite cases. Thus, there should not exist cyclic references of predicates among rules. The extension to support cyclic references is left as a future work. Throughout the proof, we always assume the depth, breadth and number of predicates of NLM are flexible and large enough to realize the demanding rules.

Here, we only prove the realization of a *definite clause*, i.e., a Horn clause with exactly one positive literal and a non-zero number of negative literals in FOL [†]. It can be written in the implication form is $\hat{p} \leftarrow p_1 \wedge p_2 \wedge \cdots \wedge p_k$ (variables as arguments are implicitly universally quantified), where $\hat{p}$ is called the *head predicate* and $p_1, \ldots, p_k$ are called *body predicates*. We group the variables appearing in the rule into three subsets: (1) variables that only appear in the head predicate, (2) variables that appear in the body predicates, and (3) variables that appear in both head and body predicates.

Consider as an example a chain-like rule: $\forall x_1 \forall x_2 \forall x_3 \forall x_4 \;\; \hat{p}(x_1, x_3, x_4) \leftarrow p_1(x_1, x_2) \wedge p_2(x_2, x_3)$. We rewrite the rule by classifying the variables:

$$\forall x_4 \Big(\forall x_1 \forall x_3 \; (\hat{p}(x_1, x_3, x_4) \leftarrow \exists x_2 \; p_1(x_1, x_2) \wedge p_2(x_2, x_3)) \Big) .$$

That is, we move all variables that ony appear in *body predicates* to the right-hand side, and extract out all variables that only appear in the *head predicate*. We show how we can compositionally combines the computation units in NLMs to realize this rule, in the following 4 steps:

[†] As for other types of Horn clauses: *facts* are realized as the tensor representations of predicates, while we implicitly views the output of NLMs as the *goal clauses*.

1. We first align the arity of the *body predicates* to include all variables that appear in at least one of the *body predicates* (including variables of set 2 and set 3). This could be done by a sequence of `Expand` operations (Eq. 2.3). In this example, we will create helper predicates to make the right-hand side of the rule as

$$\exists x_2\ p'_1(x_1, x_2, x_3) \wedge p'_2(x_2, x_3, x_1),$$

   where $p'_1(x_1, x_2, x_3) \triangleq p_1(x_1, x_2)$ and $p'_2(x_2, x_3, x_1) \triangleq p_2(x_2, x_3)$.

2. We use neural boolean logic (Eq. 2.1) to realize the boolean formula inside all quantification symbols. Moreover, we use the `Permute` operation to transpose the tensor representation so that all variables being quantified on the right-hand side appear as the last several variables in the derived predicate p'. Overall, we will derive another helper predicate,

$$p'(x_1, x_3, x_2) \triangleq p'_1(x_1, x_2, x_3) \wedge p'_2(x_2, x_3, x_1),$$

3. We use the `Reduce` operation to add quantifiers to the right-hand side (i.e., to the p' predicate). We will get:

$$p''(x_1, x_3) \triangleq \exists x_2\ p'(x_1, x_3, x_2) = \exists x_2\ p'_1(x_1, x_2, x_3) \wedge p'_2(x_2, x_3, x_1),$$

4. Finally, we use the `Expand` operation (Eq. 2.3] to add variables that only appear in the *head predicate* to the derived predicate:

$$\hat{p}(x_1, x_3, x_4) \triangleq p''(x_1, x_3).$$

   Note that, all variables appeared in the *head predicate* are implicitly universally quantified. This is consistent with our setting, since all rules in NLMs are lifted.

Overall, a symbolic rule written as a Horn clause can be realized by NLMs as a computation flow which starts from multiple expansions followed by a neural boolean rule and multiple reductions, and ends with a set of expansions.

Next, we show that the forward propagation of NLMs realizes the forward chaining of a set of Horn clauses. Following the notation in Evans and Grefenstette [2018], the forward chaining starts from a set of initial *facts*, which are essentially the grounding of base predicates. The forward chaining process sequentially applies rules over the *fact* set, and concludes new *facts*. In NLM, we represent facts as the $\mathcal{U}$-grounding of predicates.

If we consider a set of rules that does not have recursive references, all rules can be sorted in an topological order $\mathcal{R} = (r_1, r_2, \ldots, r_k)$. We only allow references of r_i from r_j, where $i < j$. Without loss of generality, we assume that the grounding of r_k is of interest. Given the topologically resolved set of rules $\mathcal{R}$, we build a set of NLMs where each NLM realizes a specific rule r_i. By stacking the NLMs sequentially, we can conclude r_k. As a side note, for multiple rules referring to the same *head predicate* $\hat{p}$, they implicitly indicate the logical disjunction ($\vee$) of the rules. We can rename these head predicates as $\hat{p}_1, \hat{p}_2, \cdots$, and use an extra NLM to implement the logical disjunction of all $\hat{p}_i$'s.

A.5 Sample Blocks World Rules

This example shows a complex reasoning in the seemingly simple Blocks World domain, which can be solved by our NLMs but requires great efforts of create manual rules by *human experts* in contrast.

Suppose we are interested in knowing *whether a block should be moved* in order to reach the target configuration. Here, a block should be moved if (1) it is movable; and (2) there is at least one block below it that does not match the target configuration. Call the desired predicate "ShouldMove(x)".

Input Relations. SameWorldID, SmallerWorldID, LargerWorldID; SameID, SmallerID, LargerID; Left, SameX, Right, Below, SameY, Above. The relations are given on all pairs of objects across both worlds.

Here is one way to produce the desired predicate by defining several helper predicates, designed by "human experts":

1. IsGround(x) $\leftarrow \forall$ y Above(y, x)
2. SameXAbove(x, y) $\leftarrow$SameWorldID(x, y) $\wedge$ SameX(x, y) $\wedge$ Above(x, y)
3. Clear(x) $\leftarrow \forall$ y $\neg$ SameXAbove(y, x)
4. Moveable(x) $\leftarrow$Clear(x) $\wedge \neg$ IsGround(x)
5. InitialWorld(x) $\leftarrow \forall$ y $\neg$ SmallerWorldID(y, x)
6. Match(x, y) $\leftarrow \neg$ SameWorldID(x, y) $\wedge$ SameID(x, y) $\wedge$ SameX(x, y) $\wedge$ SameY(x, y)
7. Matched(x) $\leftarrow \exists$ y Match(x, y)
8. HaveUnmatchedBelow(x) $\leftarrow \exists$ y SameXAbove(x, y) $\wedge \neg$ Matched(y)

9. ShouldMove(x) ←InitialWorld(x) ∧ Moveable(x) ∧ HaveUnmatchedBelow(x)

We can also write the logic forms in one line: ShouldMove(x) ←(∀ y ¬ SmallerWorldID(y, x)) ∧ (∀ y ¬ (SameWorldID(y, x) ∧ SameX(y, x) ∧ Above(y, x))) ∧ ¬ (∀ y Above(y, x)) ∧ ((∃ y SameWorldID(x, y) ∧ SameX(x, y) ∧ Above(x, y)) ∧ ¬ (∃ z ¬ SameWorldID(y, z) ∧ SameID(y, z) ∧ SameX(y, z) ∧ SameY(y, z))).

Note that this is only a part of the logic rules needed to complete the Blocks World challenge. The learner also needs to figure out where the block should be moved to. The proposed NLM can learn policies that solve the Blocks World from the sparse reward signal, indicating only whether the agent has finished the game. More importantly, the learned policy generalizes well to larger instances (consisting of more blocks).

A.6 Implement NLM in TensorFlow

The following python code contains a minimal implementation for one Neural Logic Machines layer with breadth equals 3 in TensorFlow. The `neural_logic_layer_breath3` is the main function. The syntax is highlighted and is best viewed in color.

```
from itertools import permutations
import tensorflow as tf
from tensorflow.layers import dense

def expand(input, M):
  """Expands input at its second last dimension (e.g., [B, ..., Ni,
   ↪  Nj] to [B, ..., Ni, M, Nj]) by replicating tensors."""
  ndims = input.get_shape().ndims + 1
  multiples = [M if i == ndims - 2 else 1 for i in range(ndims)]
  return tf.tile(tf.expand_dims(input, -2), multiples)

def reduce(input, M):
  """Reduces max and min at the second last dimension, except for
   ↪  diagonal elements."""
  mask = _reduce_mask(input, M)[tf.newaxis, ..., tf.newaxis]
  return tf.concat([
    tf.reduce_max(input * mask, -2),
    tf.reduce_min(input * mask + (1 - mask), -2)
```

```
  ], -1)

def neural_logic(input, hidden_dim):
  """An MLP layer applied on permutations of the input."""
  return dense(_input_permutations(input), hidden_dim,
   ↪  activation=tf.sigmoid)

def neural_logic_layer_breath3(input0, input1, input2, input3, M,
 ↪  hidden_dim, residual):
  """A neural logic layer with breath 3.
  Args:
    input0: float Tensor of shape [B, hidden_dim], nullary
     ↪  predicates.
    input1: float Tensor of shape [B, M, hidden_dim], unary
     ↪  predicates.
    input2: float Tensor of shape [B, M, M, hidden_dim], binary
     ↪  predicates.
    input3: float Tensor of shape [B, M, M, M, hidden_dim], tenary
     ↪  predicates.
    M: int, number of objects.
    hidden_dim: int, hidden dimension.
    residual: boolean, use the residual link or not.
  Returns:
    4 float Tensors, output nullary, unary, binary tenary predicates
     ↪  respectively.
  """
  agg0 = tf.concat([input0, reduce(input1, M)], -1)
  agg1 = tf.concat([input1, expand(input0, M), reduce(input2, M)],
   ↪  -1)
  agg2 = tf.concat([input2, expand(input1, M), reduce(input3, M)],
   ↪  -1)
  agg3 = tf.concat([input3, expand(input2, M)], -1)
  outputs = [neural_logic(x, hidden_dim) for x in [agg0, agg1, agg2,
   ↪  agg3]]
  if residual:
    outputs = [tf.concat([x, y], -1) for x, y in zip(outputs,
     ↪  [input0, input1, input2, input3])]
```

```
  return outputs

def _reduce_mask(input, M):
  dimension = input.get_shape().ndims - 2
  base = 1.0 - tf.eye(M)
  if dimension < 2: return tf.constant(1.0)  # Identity.
  elif dimension == 2: return base # Diagonal excluded.
  elif dimension == 3: return tf.expand_dims(base, 2) *
   ↪  tf.expand_dims(base, 1) * tf.expand_dims(base, 0)  # Mask out
   ↪  all tuples (x, y, z) that x == y or y == z or z == x.
  else: raise NotImplementedError()

def _input_permutations(input):
  dimension = input.get_shape().ndims - 2
  if dimension < 2: return input
  else: return tf.concat([
    tf.transpose(input, [0] + list(perm) + [1 + dimension])
    for perm in permutations(range(1, 1 + dimension))
  ], -1)
```

Appendix B

Appendix for Sparse and Local Neural Logic Machines

This chapter is organized as follows. First, we provide the experimental configurations and hyperparameters in Appendix B.1. Second, we provide the standard errors of SpaLoc's results in Appendix B.2. In Appendix B.3, we provide implementation details of the subgraph samplers we used. Next, we provide the ablation study on sparsification loss and SpaLoc's maximum arity in Appendix B.4, Appendix B.5, and Appendix B.6, respectively. Besides, we elaborate the calculation of information sufficiency in Appendix B.7. Finally, we define relations in the Family Tree reasoning benchmark in Appendix B.8.

B.1 Experimental configuration

Table B.1: Hyper-parameters for SpaLoc.

	Tasks	Depth	Breadth	Hidden Dims	Batch size	Subgraph size	τ
Family Tree	`HasFather`	5	3	8	8	20	1
	`HasSister`	5	3	8	8	20	2
	`Grandparent`	5	3	8	8	20	2
	`Uncle`	5	3	8	8	20	2
	`MGUncle`	5	3	8	8	20	2
	`Family-of-three`	5	3	8	8	20	2
	`Three-generations`	5	3	8	8	20	2
Inductive KG	`WN18RR`	6	3	64	128	10	3
	`FB15K-237`	6	3	64	64	20	3
	`NELL-995`	6	3	64	128	10	3
Transductive KG	`WN18RR`	6	3	64	64	20	3
	`FB15K-237`	6	3	64	64	20	3
	`NELL-995`	6	3	64	64	20	3

Table B.2: SpaLoc's results with standard errors on real-world knowledge graph inductive reasoning datasets from GraIL (Table 2.6).

WN18RR				FB15k-237				NELL-995			
v1	v2	v3	v4	v1	v2	v3	v4	v1	v2	v3	v4
$98.45_{\pm 0.20}$	$99.96_{\pm 0.04}$	$96.63_{\pm 0.05}$	$99.34_{\pm 0.01}$	$99.74_{\pm 0.02}$	$99.34_{\pm 0.03}$	$99.55_{\pm 0.01}$	$99.29_{\pm 0.08}$	$100_{\pm 0.00}$	$98.56_{\pm 0.00}$	$97.19_{\pm 0.08}$	$97.34_{\pm 0.06}$

Table B.3: SpaLoc's results with standard errors of transductive link prediction on real-world knowledge graphs (Table 2.7).

WN18RR		NELL-995		FB15K-237	
AUC-PR	Hit@10	AUC-PR	Hit@10	AUC-PR	Hit@10
$96.74_{\pm 0.02}$	$99.94_{\pm 0.03}$	$99.25_{\pm 0.04}$	$98.93_{\pm 0.04}$	$99.69_{\pm 0.05}$	$96.96_{\pm 0.04}$

We optimize all models with Adam [Kingma and Ba, 2015] and use an initial learning rate of 0.005. All experiments are under the supervised learning setup; we use Softmax-Cross-Entropy as the loss function.

Table B.1 shows hyper-parameters used by SpaLoc. For all MLP inside SpaLoc, we use no hidden layer and the sigmoid activation. Across all experiments in this paper, the maximum arity of intermediate predicates (i.e., the "breadth") is set to 3 as a hyperparameter, which allows SpaLoc to realize all first-order logic (FOL) formulas with at most three variables, such as a "transitive relation rule." We set the specification threshold ϵ to 0.05 in our experiments. This value is chosen based on the validation accuracy of our model. In practice, we observe that any values between 0.01 and 0.1 do not significantly impact the performance of our method and the inference complexity. We also set the multiplier of the sparsification loss λ to 0.01 in all SpaLoc's experiments.

B.2 SpaLoc's results with standard errors

We show the standard errors of SpaLoc's results in Table B.2 and Table B.3. The scores and standard errors are obtained from experiments with three different random seeds. We find the standard erros are small, i.e., the performance of SpaLoc is quite stable.

B.3 Implementation of subgraph samplers

Both the neighbor expansion and the path sampler sample subgraphs by inducing from selected node sets. To deal with input hypergraphs with any arities, the samplers simplify the input hypergraphs into binary graphs. We define two nodes in the hypergraph are connected

Table B.4: Comparison of different sparsification losses.

	HasSister		Grandparent		Uncle	
	Accuracy	Density	Accuracy	Density	Accurcay	Density
L_1	91.81	0.48%	99.8%	0.99%	74.69%	0.68%
L_2	100	0.75%	100%	0.61%	94.46%	2.44%
H_S	100	0.51%	100%	0.48%	100%	0.87%

Table B.5: Comparison (Per-class Accuracy) of SpaLoc with different max arities on family tree reasoning benchmarks.

	HasSister	Grandparent	Uncle
Max Arity = 2	87.66	86.74	50.00
Max Arity = 3	**100**	**100**	**100**

if they are covered by a hyperedge. Therefore, the neighbor expansion and path-finding algorithms used by the samplers can be applied to any hypergraphs for finding node sets. After enough nodes are collected, the sampler will induce a sub-hypergraph from the original hypergraph by preserving all of the hyperedges lying in the set.

B.4 Ablation study on sparsification loss

We compare our Hoyer-Square sparsification loss against the L_1 and L_2 regularizers on the family tree datasets. In Table B.4, we show the performance of SpaLoc trained with different sparsification losses. All models are tested on domains with 100 objects.

"Density" is the percentage of non-zero elements (NNZ) in the intermediate groundings. The lower the density, the better the sparsification and the lower the memory complexity. We can see that, compared with L_1 and L_2 regularizers, the Hoyer-Square loss yields a higher or comparable sparsity while maintaining nearly perfect accuracy.

B.5 Ablation study on SpaLoc's maximum arity

In this section, we compare two different SpaLoc models with different maximum arity, to validate the necessity and effectiveness of hyperedges in inductive reasoning. Shown in Table B.5 and Table B.6, we compare SpaLoc models with max arity 2 and 3. Note that, even if the input and output relations are binary, adding ternary edges in the intermediate representations significantly improves the result.

Table B.6: Comparison (AUC-PR) of SpaLoc with different max arities on real-world knowledge graph inductive relation prediction task.

	WN18RR-v1	FB15k-237-v1	NELL-995-v1
Max Arity = 2	97.65	90.71	94.58
Max Arity = 3	**98.18**	**99.73**	**100**

B.6 Ablation Study on Label Adjustment

Table B.7: Comparison (per-class accuracy) for different label calibration methods.

Sampler	`HasFather`			`HasSister`		
	NC	LS	IS	NC	LS	IS
Node	50.00	50.00	50.00	50.00	50.00	**80.72**
Walk	50.00	52.41	**100**	59.90	75.13	**93.16**
Neighbor	50.00	51.63	**100**	75.29	78.06	**98.01**

In this section, we compare our information sufficiency-based label adjustment method (IS) against two simple baselines: no adjustment ("NC"), and label smoothing (LS). In "LS", we multiply all positive labels with a constant $\alpha = 0.9$.

Results are shown in Table B.7. Overall, our method (IS) outperforms both baselines, even when the average information sufficiency of training graphs is very low (e.g., when using the *Node* sampler). Especially on the *HasFather* task, using constant label smoothing only has a close-to-chance accuracy (50%). Combining our IS-based label adjustment with the *Neighbor* sampler yields the best result.

B.7 Calculation of the information sufficiency

The crucial part in the computation of the information sufficiency is to count the number of k-hop hyperpaths connecting a given set of nodes $\{v_1, \ldots, v_r\}$ in a hypergraph. We use the incidence matrix to calculate this. Firstly, we use a $n \times m$ incidence matrix B to represent the hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$, such that $B_{ij} = 1$ if the vertex v_i and edge e_j are incident and 0 otherwise. Next, $B^{(k)} := (BB^T)^{k-1}B$ is the k-hop incidence matrix of the hypergraph, i.e., $B^{(k)}_{ij}$ is the number of k-hop paths that the vertex v_i and edge e_j are incident.

For example, when $r = 2$, there are $B^{(k)}_i B^T_j$ k-hop paths connecting vertex v_i and v_j. When $r = 1$, there are $\sum_j B^{(k)}_{ij}$ k-hop paths connecting to vertex v_i.

B.8 Definition of Relations in Family Tree

The inputs predicates are: $\texttt{Father}(x, y)$, $\texttt{Mother}(x, y)$, $\texttt{Son}(x, y)$, $\texttt{Daughter}(x, y)$.

The target predicates are:

- $\texttt{HasFather}(x) := \exists a\ \texttt{Father}(x, a)$
- $\texttt{HasSister}(x) := \exists a \exists b\ \texttt{Father}(x, a) \wedge \texttt{Daughter}(a, b) \wedge \neg(b = x)$
- $\texttt{Grandparent}(x, y) := \exists a\ \texttt{parent}(x, a) \wedge \texttt{parent}(a, y)$

  $\texttt{parent}(x, y) := \texttt{Father}(x, y) \vee \texttt{Mother}(x, y)$
- $\texttt{Uncle}(x, y) := \exists a\ \texttt{Grandparent}(x, a) \wedge \texttt{Son}(a, y) \wedge \neg\texttt{Father}(x, y)$
- $\texttt{MGUncle}(x, y) := \exists a \exists b\ \texttt{Grandmother}(x, a) \wedge \texttt{Mother}(a, b) \wedge \texttt{Son}(b, y)$

  $\texttt{Grandmother}(x, y) = \exists a\ \texttt{Parent}(x, a) \wedge \texttt{Mother}(a, y)$
- $\texttt{Family-of-three}(x, y, z) = \texttt{Father}(x, y) \wedge \texttt{Mother}(x, z)$
- $\texttt{Three-generations}(x, y, z) = \texttt{Parent}(x, y) \wedge \texttt{Parent}(y, z)$

We follow the dataset generation algorithm presented in Chapter 2. In detail, we simulate the growth of families to generate examples. For each new family member, we randomly assign gender and a pair of parents (can be none, which means it is the oldest person in the family tree) for it. After generating the family tree, we label the relationships according to the definitions above.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix C

Appendix for Temporal and Object Quantification Networks

C.1 Implementation Details

In this section, we present the implementation detail of our model, the TOQ-Nets, and five baselines (STGCN, STGCN-LSTM, STGCN-MAX, Space-Time and Non-local), including the model architecture, input features, and the training methods.

C.1.1 TOQ-Nets

In the soccer event detection task, we use three object quantification layers and three temporal quantification layers. Each object quantification layer has a hidden dimension of 16 (i.e., all nullary, unary, and binary tensors have the same hidden dimension of 16). Each temporal quantification layer has a hidden dimension of 48.

In the manipulation concept learning task, we use three object quantification layers and three temporal quantification layers. Each object quantification layer has a hidden dimension of 48. Each temporal quantification layer has a hidden dimension of 144.

Input feature extractor. We use the following physics-based input feature extractor for soccer event detection. Given the trajectories of all entities, including all players and the ball, we first compute the following physical properties: ground speed (i.e., the velocity on the xy-plane), vertical speed (i.e., the velocity along the z direction), and acceleration. We also compute the distance between each pair of entities.

After extracting the physical properties, we use the following feature templates to generate the input features. For each physical property, X, we first normalize X across all

training trajectories, and then we create $c = 5$ features:

$$\sigma\left(\frac{X - \theta_i}{\beta}\right),$$

where σ is the sigmoid function, β is a scalar hyperparameter, $\theta_i, i = 1, 2, 3, 4, 5$ are trainable parameters. These feature templates can be interpreted as differentiable implementations of $\mathbf{1}[X > \theta_i]$, where $\mathbf{1}[\cdot]$ is the indicator function. During training, we make β decay exponentially from 1 to 0.001.

In the manipulation concept learning task, we use a single fully-connected layer with sigmoid activation as the feature extractor. The hidden dimension of the layer is 64.

C.1.2 STGCN

We use the same architecture as Yan *et al.* [2018]. Table C.4 summarizes the hidden dimensions and the kernel sizes.

The STGCN model's output is a tensor of size $(T/4) \times N \times 256$, where T is the length of the input trajectory, and N is the number of entities. Following Yan *et al.* [2018], we perform an average pooling over the temporal and the entity dimension and get a feature vector of size 256. We apply a softmax layer on top of the latent feature to classify the action.

Input feature extractor. In the soccer event detection task, we do not input the distance between each pair of entities as we do for the TOQ-Nets, as the STGCN model does not support binary predicate inputs (see STGCN-2D for an ablation study). In the manipulation concept learning task, STGCN uses the same input format as the TOQ-Nets. Specifically, we represent the state of each entity with 13 unary predicates, including 3D position, 3D orientation, and bounding boxes.

C.1.3 STGCN-MAX

The STGCN-MAX model is a variation of STGCN. During graph propogation, for each node, instead of taking average of all propagated information from all neighbors, we take their maximum value over all feature dimensions, and update the hidden state of each node with this. This baseline replicates a very similar propagation rule as our relational reasoning layer. However, it still uses temporal convolutions to model temporal structures.

	Reg.	New	All
TOQ-Net	**87.7**	**52.2**	**79.8**
STGCN-2D	84.1	46.9	75.8

Table C.1: **Ablation study of the STGCN-2D model on the soccer event dataset** Models are evaluated by their few-shot learning performance.

	9-way	3v3	11v11	6v6(Time warp)
TOQ-Net	**88.4**	**77.4**	**77.1**	**86.9**
STGCN-2D	84.5	17.5	16.6	39.7

Table C.2: **Ablation study of the STGCN-2D model on the soccer event dataset** Models are evaluated by their generalization to different numbers of players and time-warped trajectories.

C.1.4 STGCN-2D

The original STGCN does not support input features of higher dimensions, so we extend STGCN to STGCN-2D to add binary inputs such as the distance. This extension slightly improves the model performance on the standard 9-way classification test. It also helps in the few-shot setting. Specifically, on the few-shot learning setting:

Meanwhile, adding extra binary inputs does not improve the generalization of the network to games with a different number of agents or to time-warped sequences. This following table extends the Table 2 in the main text. Specifically, we train STGCN-2D on 6v6 soccer games and test it on scenarios with a different number of agents (3v3 and 11v11) and also temporally warped trajectories. Our model shows a significant advantage.

C.1.5 STGCN-LSTM

The STGCN-LSTM model first encodes the input trajectory with an STGCN module. The output of the STGCN module is a tensor of shape $T \times N \times 256$, where T is the length of the input trajectory, and N is the number of entities. We perform an average pooling over the entity dimension and get a tensor of shape $T \times 256$. Next, we encode this feature with a 2-layer bidirectional LSTM module. The hidden dimension for the LSTM module is 256. We use a single softmax layer on top of the LSTM module to classify the action.

STGCN-LSTM uses the same architecture as STGCN for both tasks, except for the kernel size and the stride, because our goal is to evaluate the performance of recurrent neural networks (the LSTM model) on modeling temporal structures. In the soccer event detection task, we use kernel size 3 and stride 1 so that the STGCN module will have the

Model	Reg.	1-Shot	Full
STGCN	$99.89_{\pm 0.05}$	$94.92_{\pm 1.03}$	$98.79_{\pm 0.23}$
STGCN-LSTM (kernel size 3)	$99.89_{\pm 0.05}$	$96.16_{\pm 1.39}$	$99.04_{\pm 0.31}$
STGCN-LSTM (kernel size 1)	$99.92_{\pm 0.03}$	$95.48_{\pm 1.67}$	$98.86_{\pm 0.43}$
TOQ-Net	$\mathbf{99.96}_{\pm 0.02}$	$\mathbf{98.04}_{\pm 0.97}$	$\mathbf{99.48}_{\pm 0.24}$

Table C.3: **Ablation study of different kernel sizes for the STGCN-LSTM model on the RLBench dataset** Performance is measured by per-action (macro) accuracy and averaged of four 1-shot splits and four random seeds per split. The ± values indicate standard errors.

representational power to encode local physical properties. In the manipulation concept learning task, we use a temporal kernel size of 1 and stride 1. Meanwhile, Table C.3 shows the ablation study of different kernel sizes, evaluated by the few-shot learning performance on the RLBench dataset. Different kernel sizes (1 and 3) have a similar performance.

C.1.6 Space-Time Region Graph Networks

We use the same architecture for each Space-Time layer as Wang et al. proposed in Space-Time Graph [Wang and Gupta, 2018]. Table C.5 summarises the detailed parameters of all layers.

C.1.7 Non-Local Neural Networks

We use the same architecture for each Non-local layer as Wang et al. proposed in Non-Local networks [Wang *et al.*, 2018b]. The original non-local network works on pixels. Here instead, we apply it to the space-time graph constructed in the same way as Wang and Gupta [2018]. The major difference between the Space-Time Graph and the Non-local Network is that Space-Time Graph uses graph convolutions while Non-local uses transfomer-like attentions over all nodes in the graph. Table C.6 summarises the detailed parameters of all layers.

Soccer event detection. We use the identical training strategy for our model and both baselines. Each training batch contains 128 examples, which are sampled as following: we first uniformly sample an action category, and then uniformly sample a trajectory labeled as this action category.

In all the soccer event tasks, we optimized our model using Adam [Kingma and Ba, 2015], with the learning rate initialized to $\eta = 0.002$. The learning rate is fixed for the first

Model	Input Dim.	Output Dim.	Kernel Size	Stride	Dropout	Residual
STGCN	-	64	7	1	0	False
	64	64	7	1	0.5	True
	64	64	7	1	0.5	True
	64	64	7	1	0.5	True
	64	128	7	2	0.5	True
	128	128	7	1	0.5	True
	128	128	7	1	0.5	True
	128	256	7	2	0.5	True
	256	256	7	1	0.5	True
	256	256	7	1	0	True
STGCN_S	-	16	7	1	0	False
	16	16	7	1	0.5	True
	16	32	7	2	0.5	True
	32	32	7	1	0.5	True
	32	64	7	2	0.5	True
	64	128	7	1	0	True
STGCN_T	-	8	7	1	0	False
	8	12	7	2	0.5	True
	12	64	7	1	0	True

Table C.4: **The STGCN architecture used in the chapter.**

50 epochs. After that, we decrease the learning rate η by a factor of 0.9 if the best validation loss has not been updated during the last 6 epochs.

All models are trained for 200 epochs. In the task of generalization to different input scales, we first train different models on the original dataset for 200 epochs (~40,000 iterations) and finetune them on the new dataset with few samples for 10,000 iterations.

Manipulation concept learning. We optimized different models using Adam with the initial learning rate $\eta = 0.003$. We applied the same learning rate schedule as in the soccer event detection task. The batch size for the manipulation concept learning task is 32 for models with STGCN backbone and is 4 for TOQ-Nets. All models are trained for 150 epochs.

Model	Input Dim.	Output Dim.	Stride	Residual
Space-Time	-	64	1	False
	64	64	1	True
	64	64	1	True
	64	64	1	True
	64	128	2	True
	128	128	1	True
	128	128	1	True
	128	256	2	True
	256	256	1	True
	256	256	1	True
Space-Time$_S$	-	16	1	False
	16	16	1	True
	16	32	2	True
	32	64	2	True
	64	128	1	True
Space-Time$_T$	-	32	1	False
	32	64	2	True
	64	64	2	True

Table C.5: **The Space-Time Region Graph architecture used in the chapter.**

Model	Input Dim.	Output Dim.	Stride	Residual
Non-Local	-	64	1	False
	64	64	1	True
	64	64	1	True
	64	64	1	True
	64	128	2	True
	128	128	1	True
	128	128	1	True
	128	256	2	True
	256	256	1	True
	256	256	1	True
Non-Local$_S$	-	32	1	False
	32	32	2	True
	32	64	2	True
	64	128	1	True
Non-Local$_T$	-	32	1	False
	32	32	2	True
	32	64	2	True

Table C.6: **The Non-local Neural Networks architecture used in the chapter.**

Appendix D

Appendix for Expressivity of Neuro-Symbolic Logic Models

This chapter is organized as the following. In Appendix D.1, we provide the proof details and more discussion with related work on search complexity. In Appendix D.2, we provide the proof details and more discussion on policy circuit complexity. Next, in Appendix D.3, we include a detailed analysis of regression widths and circuit complexities for empirical planning problems. Finally, in Appendix D.4, we describe environmental setups and implementation details for the experiments.

D.1 Proofs and Discussions for Search Complexity

D.1.1 Complete and Optimal Version of S-GRS

Algorithm 11 shows a complete and optimal version of the serialized goal regression search algorithm. The key difference between this variant and the simplified one presented in the paper is that now we keep track of multiple possible trajectories that can achieve the specified goal. The algorithm is complete because for any set of precondition atoms, in the optimal trajectory, there will always be an order in which they have been achieved (note that some of the sub-trajectories can be empty, indicating that while planning for the prefix the next subgoal has already been achieved; this covers the "parallel" precondition achievement case.). We show an example of this branching factor in Figure D-1. Note that, to make sure that the algorithm eventually terminates, we need to make sure that all trajectories do not have loops.

However, this bare algorithm can be very slow: the worst time complexity of it is $O(3^N)$ with respect to N, the number of atoms in $\mathcal{P}_0$ — in contrast to $O(2^N)$ for a simpler backward

search algorithm*. This is because the possible state set for different subgoals and different constraints may overlap, so it will "waste" time during the search. Therefore, this algorithm may not be of practical use.

Algorithm 11 Serialized goal regression search with multiple optimal path tracking.

function GRSOPT(s_0, g, *cons*)
  if $g \in s_0$ **then return** (s_0)
  all_possible_t = empty_set()
  for $r \in \mathcal{R}_0$: $cons \cap eff_(action(r)) = \emptyset \wedge goal(r) = g$ **do**
    $p_1, p_2, \cdots, p_k$ = subgoals(r)
    if $\exists p_i$ s.t. $p_i \in$ *goal stack* **then**
      continue
    possible_t = $\{\emptyset\}$
    for i in $1, 2, \cdots, k$ **do**
      next_possible_t = empty_set()
      for *prefix_t* in *possible_t* **do**
        imm_state = *last_state*(*prefix_t*)
        for $nt \in$ GRSOPT(*imm_state*, p_i, $cons \cup \{p_1, \cdots, p_{i-1}\}$) **do**
          next_possible_t = *next_possible_t* $\cup$ {*prefix_t* + nt}
      possible_t = *next_possible_t*
    all_possible_t = *all_possible_t* $\cup \{t + \mathcal{T}(last_state(t), action(r)) \mid t \in possible_t\}$
  return *all_possible_t*

D.1.2 Proof for Theorem 4.3.1: S-GRS Optimality and Completeness

We define the set of optimally serializable rules for a state s as $OSR(s)$, and the set of single-literal goals that can be solved with $OSR(\cdot)$ from s_0 as $OSG(s_0)$.

Proof. Completeness: the definition of $OSG(s_0)$ asserts that g can be solved by S-GRS.

Optimality: proof by induction. In the following, we prove that for any $g \in OSG(s_0)$, the trajectory returned by S-GRS is optimal.

First, we consider the set of all S-GRS calls in the regression tree that contributes to the returned trajectory. We do induction based on a distance metric. Consider any goal atoms $g \in OSG(s_0)$, let $Distance(s_0, g, cons)$ be the length of the path returned by S-GRS. First, for all argument tuples with distance 0 (i.e., $g \in s_0$ and $cons \subseteq s_0$), S-GRS will return the optimal trajectory.

Next, assuming that S-GRS will return optimal trajectories for all argument tuples with distances strictly smaller than d. Consider a new argument call $grs(s_0, goal, cons)$ with distance d. Consider now all optimally serializable and goal-achieving rules $r \in OSR(s_0)$

*Note that *constraints* is always a subset of state *state*, so the worst complexity is $O(3^N)$.

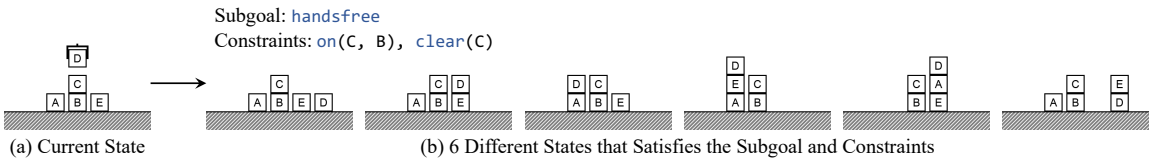


Figure D-1: **Illustration of the branching factor caused by tracking multiple resulting states after achieving a subgoal.**

such that $goal(r) = g$ and $cons \cap eff_(action(r)) = \emptyset$. Let r_g be the rule that returns the plan during search, and $p_1 \cdots p_k$ be the serialized precondition list. First, all sub-function calls to S-GRS will return optimal plans by induction. Second, by optimal serializability of the rule r, we know that concatenating the sub-plans will yield an optimal plan to achieve g while preserving g. Therefore, our induction holds. □

D.1.3 Making Non-SOS Problem Serializable

Here we discuss a general strategy to make non-SOS problems serializable by introducing super predicates. In particular, for a given operator, if two of the preconditions (without loss of generality, say, $p(x)$ and $q(x)$) are not serializable, we can introduce a new predicate $p_and_q(x)$, and rewrite the precondition of the operator with this new super predicate.

The disadvantage of this encoding is that it will enlarge not only the predicate set but also the operator set. In particular, for any operator o such that $p(x) \in eff_+(o)$ or $p(x) \in eff_(o)$, we need to break the operator into two sub-operators. For example, if $p(x) \in eff_+(o)$, we need to create a new operator o_1 that has $\neg q(x)$ in its precondition, and set $eff_+(o_1) = eff_+(o)$ and $eff_(o_1) = eff_+(o)$. For the second operator o_2, it should have $q(x)$ in its precondition and $eff_+(o_1) = eff_+(o) \cup \{p_and_q(x)\}$ and $eff_(o_1) = eff_(o)$. As the number of predicates in this super predicate increases, the number of additional operators grows exponentially.

This provides a characterization of "what contributes to the (forward) width of a problem." Specifically, we have identified two sources of forward widths: non-serializability of certain operators, and the need to track constraints while doing a regression search.

D.1.4 Proof for Theorem 4.3.2: Polynomial Hardness for SOS-k problems.

If a problem P is of SOS width k, it can be solved in time $O(N^{k+1})$, where N is the number of atoms. Here we have omitted polynomials related to enumerating all possible actions and their serializations (which are polynomial w.r.t. the number of objects in $\mathcal{U}$).

Proof. First, if we are given the rule set $\mathcal{R}$, then the proof can be simply done by analyzing

function calls: it is important to observe that there are at most $O(N^{k+1})$ possible argument combinations of $(goal, cons)$ to function GRS.

First, due to SOS condition, we only need to maintain one single optimal trajectory for each set of argument values $(s_0, goal, cons)$. Next, we consider we only maintain one single optimal trajectory for each pair of $(goal, cons)$, ignoring the first argument s_0. Note that, even if the "cached" trajectory does not include the input state s_0 we can still return the cached trajectory. This is because the optimal serializability suggests that any trajectories for achieving $(goal, cons)$ can be extended to future preconditions.

Finally, we consider removing the assumption of knowing $\mathcal{R}$. Note that the enumeration of all possible goal width-k regression rules can be done in $O(2^k)$, which is considered constant when k is small. Therefore, overall the algorithm runs in polynomial time and the solver does not need to know the set of optimally serializable regression rules $\mathcal{R}$ a priori. □

First, the fact that the algorithm does not need to know $\mathcal{R}$ *a priori* is important because it suggests that, as long as there exists a set of optimally serializable rules for the problem we are trying to solve, the search algorithm will find a solution, without having any additional knowledge other than the operator definitions. Intuitively, this is because doing S-GRS with additional non-optimally serializable rules (e.g., using all rules in $\mathcal{R}_0$) will not falsify the completeness and optimality of the algorithm.

Second, unfortunately, the SOS condition can not be relaxed. That is, if we only assume optimal serializability of generalized regression rules and all rules have width k, the proof will fail. This is because even if regression rules have a small width, there will be a possibly exponential branching factor caused by "free variables" in regression rules that do not appear in the goal atom. Concretely, let us consider the following rule $g(x) \leftsquigarrow p_1(x, y)^{\emptyset}, p_2(x)^{\emptyset} \parallel a_1(x, y)$. This rule is of width 0; however, since there is no guarantee that any optimal trajectory for achieving $p_2(x)$ will also achieve $p_1(x, y) \wedge p_2(x)$, during search, we can not use the cached trajectory for $(p_2(x), \emptyset)$ as the return value when we solve for $p_2(x)$ after achieving $p_1(x, y)$, which breaks the complexity analysis.

D.1.5 Additional Discussion

Deriving optimally-serializable, small-width rules. For a practical system, if we can know the set of regression rules needed, we can accelerate the search significantly. Here we discuss a strategy for deriving such rules by analyzing the domain definition itself.

If we ignore the optimality, a sufficient condition that a rule r is not strong optimal serializability for a given state s_0 is that, there exists a precondition p_i such that there is no rule r' such that $goal(r') = p_i$, $eff_(action(r')) \cap p_i = \emptyset$ and $goal(r) \notin subgoals(r')$. The last condition is the goal-stack checking: if achieving a goal g using rule r requires achieving

goal g itself, then the rule is not serializable.

Putting this into practice, consider all rules for achieving *clear*(A). If there is another object B on A in s_0, then only the rule derived from *unstack*(B, A) will be SOS serializable. This is because, for any other rules *unstack*(y, A), where $y \neq B$, achieving *on*(y, A) will involve achieving *clear*(A) itself. Note that this analysis can be done by static (lifted) analysis of the domain.

This analysis can be viewed as simulating a one-step goal regression search in a lifted way, and it can be extended to more steps. In some domains, this strategy will help rule out some serialization of the preconditions (such as the BlocksWorld example), but in general, whether a rule is strong optimally serializable is state-dependent and can be hard to compute.

Connection with other concepts in search. Another view of the regression width is the maximum number of constraints one needs to track while applying S-GRS recursively. This notion is analogous to the treewidth of a constraint graph in constraint satisfiability problems (CSPs): intuitively, the number of backward-pointing edges [Freuder, 1982].

As also briefly discussed in Lipovetzky and Geffner [2012], delete-relaxation heuristics are also related. In particular, the hFF heuristic can be interpreted as applying backward search by considering "parallelizability" in contrast to "serializability." Informally, for a given state s and a sequence of preconditions p_i, we compute trajectories for each individual p_i separately and concatenate them.

Generalization to ∀-quantified preconditions. It would be possible to extend the definition to ∀-quantified preconditions by simply allowing regression rules to have an unbounded number of subgoals (e.g., to handle goals such as ∀*xon-table*(x)). However, this is not helpful in general because the constraints will accumulate as the number of subgoals becomes larger, and enumerating subgoal serialization will now take $O(m!)$ where m is the number of subgoals. This notion will be more helpful when we have a "goal regression rule selector."

D.1.6 Connection with Width-Based Forward Search IW(k)

Wecall the width of a problem defined in Chen and Giménez [2007] and the Iterated Width algorithm $IW(k)$ [Lipovetzky and Geffner, 2012] as forward width. Then we have:

Theorem D.1.1 (Sufficiency condition for forward width)**.** *Any planning problem that has SOS-width k has a forward width of at most* $k + 1$ *and, hence, can be solved by the algorithm* $IW(k + 1)$.

Proof. We prove that for any trajectory returned by the S-GRS algorithm for an SOS width-k problem P, all actions in the trajectory achieve a novel $k + 1$-sized atom set. If this is true,

then the final goal will be reachable from s_0 in the size $k+1$ tuple graph in IW search.

Consider any action a in the returned plan, it corresponds to a unique rule for achieving a subgoal g and a set of constraints *cons*. If the size $k+1$ tuple $\{g\} \cup cons$ has already been achieved before a, then due to the SOS condition, we can make the returned plan shorter. This violates the optimality of the S-GRS algorithm. □

Unfortunately, these two definitions are not exactly equivalent to each other. For example, consider the Blocks World domain and a state of three stacked blocks A, B and C, the goal regression rule

$$clear(\mathrm{A}) \leftsquigarrow on(\mathrm{B}, \mathrm{A}), clear(\mathrm{B})^{\emptyset}, handsfree()^{\{clear(\mathrm{B})\}} \parallel unstack(\mathrm{y}, \mathrm{x})$$

is SOS. It has width 1 because while achieving *handsfree*(), we have to explicitly maintain *clear*(B); otherwise, it is possible that we directly put C back onto B instead of putting C onto the table (both plans are optimal for achieving *handsfree*()) after achieving *clear*(B). By contrast, in IW, the Blocks World domain has width 1 instead of 2 if the goal predicate is *clear*, because the optimal plan for achieving $handsfree()^{\{clear(\mathrm{B})\}}$ happens to be the optimal plan for achieving *on-table*(C).

Proofs for (forward) widths of planning problems in IW were mostly done by analyzing solution structures. By contrast, the constructions in this paper form a new perspective to interpret planning problem width: they are subgoals and constraints that need to be satisfied during a goal regression search. This view is helpful because our analyses focus on analyzing concrete goal regression rules, and the derived results are compositional. Unfortunately, not all problems that can be solved by $IW(k)$ have a regression width of $k-1$.

D.2 Proofs and Discussions for Policy Circuit Complexity

D.2.1 Proof of Theorem 4.4.1: Compilation of BWD

Given a planning problem P, let T be the length of the optimal trajectory as (the planning horizon), k_{BWD} be the maximum number of atoms in the goal set in BWD, and β be the maximum arity of atoms in the domain. The backward search algorithm BWD can be compiled into a relational neural network in $RelNN[O(T), \beta \cdot k_{\text{BWD}}]$.

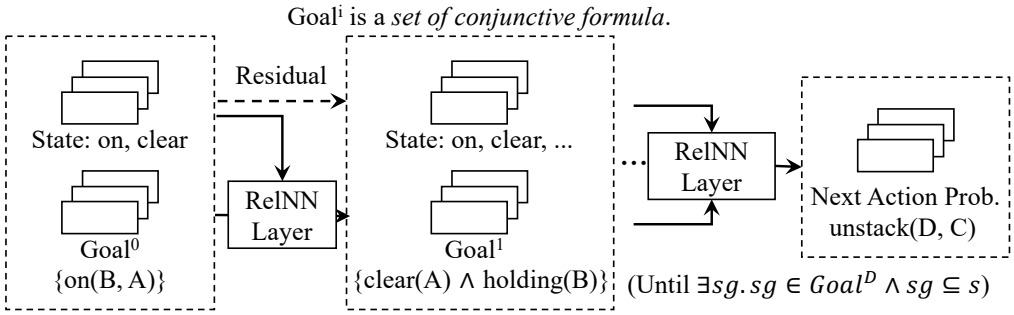


Figure D-2: **Illustration of the compilation of backward search into a *RelNN* policy.**

Proof. There is a proof by construction. As illustrated in Fig. D-2, a relational neural network can use its intermediate representations to keep track of a set of subgoals (i.e., a set of sets of ground atoms) $Goal^d$ at each layer d. Initially, $Goal^0$ contains only the goal atom. Then, $sg \in Goal^d$ if there is a path of length d from any state s that satisfies $sg \subseteq s$ to the final goal of the planning problem. For all layers d, for any subgoals sg, for all ground actions $a \in \mathcal{A}$, $sg \in Goal^d \wedge (eff_-(a) \cap sg = \emptyset) \Rightarrow (sg \cup pre(a) \setminus eff_+(a)) \in Goal^{d+1}$. Since the maximum number of atoms in BWD is k_{BWD}, we only need to keep track of ground atom conjunctions that involve at most $k_{\text{BWD}} \cdot \beta$ distinct objects. Therefore, BWD search can be compiled into a relational neural network in $RelNN[O(T), k_{\text{BWD}} \cdot \beta]$.

A more detailed construction is the following. Let $k' = k_{\text{BWD}} \cdot \beta$. Let $x_1, \cdots, x_{k'}$ be variables. $\{q_i(x_1, \cdots, x_{k'})\}$ is the set all possible conjunctive expressions that involve variables $\{x_1, \cdots, x_{k'}\}$. Let $q_i^{(d)}(x_1, \cdots, x_{k'})$ be the value of $q_i(x_1, \cdots, x_{k'})$ at layer d, and $expr(q_i)$ be the conjunctive expression of q_i. For all d, q_i, q_j, and for all actions a that are grounded on variables from $\{x_1, \cdots, x_{k'}\}$, if $(eff_-(a) \cap expr(q_i) = \emptyset) \wedge (expr(q_j) = expr(q_i) \cup pre(a) \setminus eff_+(a))$, there is a logical rule: $\forall x_1, \cdots, x_{k'}.\ q_i^{(d)}(x_1, \cdots, x_{k'}) \Rightarrow q_j^{(d+1)}(x_1, \cdots, x_{k'})$. Furthermore, for all d, q_i, if $(eff_-(a) \cap expr(q_i) = \emptyset) \wedge (expr(q_i) \cup pre(a) \setminus eff_+(a) \subset s)$, where s is the current state, we have the following rule that outputs the next action to take, $\forall x_1, \cdots, x_{k'}.\ q_i^{(d)}(x_1, \cdots, x_{k'}) \Rightarrow a^{(d+1)}$, where $a^{(d)} = 1$ indicates that there exists a length d trajectory for achieving the goal. In practice, we will the shortest one. Note that all these

rules are fully quantified by $\forall x_1, \cdots, x_{k'}$ so they are lifted FOL rules and can be implemented by a *RelNN* circuit. □

D.2.2 Proof of Theorem 4.4.2: Compilation of S-GRS

Given a planning problem P of SOS width k, let T be the length of the optimal trajectory, and β be the maximum arity of atoms in the domain. The serialized goal regression search S-GRS can be compiled into a $RelNN[O(T), (k+1)\cdot\beta]$, where $k+1 \leq k_{\text{BWD}}$.

Proof. There is a proof by construction. The easiest construction is based on the sufficient condition between regression width k and $IW(k)$. In particular, based on Theorem 4.3.3, we know that $IW(k+1)$ can solve the planning problem P. Therefore, we can use the following construction to simulate $IW(k+1)$.

The most important trick in the construction is that a relational neural network can use its intermediate representations to keep track of two sets of atoms. First, for each atom tuple t up to size $k+1$, whether t is reachable from s_0 within d steps. Second, let s_t be the last state of the optimal trajectory associated with t; for each atom tuple t up to size $k+1$ and for a predicate $p(x_1, x_2, \cdots, x_\beta)$, we keep track of whether $p(x_1, x_2, \cdots, x_\beta)$ is true in s_t. At each layer d, we consider all possible ground actions $a \in \mathcal{A}$, if there exists such (t_1, t_2) such that t_2 is not reachable within d steps and $t \subseteq \mathcal{T}(s_{t_1}, a)$, we set t_2 to be reachable within $d+1$ and $s_{t_2} = \mathcal{T}(s_{t_1}, a)$.

In addition to the realizability of the $IW(k+1)$ algorithm, we need to additionally prove that, if we follow the policy derived by simulating $IW(k+1)$ for the first step, we can recurrently apply this policy. To see that, consider the path from s_0 to the goal state in $IW(k+1)$ in the tuple graph. Since we are moving forward by one step in this tuple path, doing a search from the second tuple in the path (i.e., applying the same *RelNN* policy at the second world state) will yield the same trajectory.

Similar to the construction in Theorem 4.4.1, all the rules are lifted FOL rules and can be implemented by a *RelNN* circuit. It is also possible to use the same trick to compile the original S-GRS algorithm: essentially, for each $(g, cons)$ pair, keep track of the optimal trajectory for achieving it. However, the construction will be more complicated and is omitted here. □

D.2.3 More Discussion

For some problems, even if we have a regression rule selector, to compute a plan, we may still need an unbounded depth of search steps (i.e., d depends on the number of objects in the domain). For example, in Blocks World, to achieve *clear*(A), the number of steps

needed is proportional to the number of objects on top of A. There are two ways to build a relational neural network policy in this case. First, we can allow the network to have a variable number of iterations by applying the same layer recurrently several times. We have already illustrated this in the experiment.

Second, there might be "shortcuts" that we can learn. An intuitive example is that in BlocksWorld, the goal regression for *clear*(A) will recur until we have found the topmost block stacked on top of A. If the domain has a new predicate $above(x, y)$ indicating whether the block x is above another block y, then we can use just one single rule to predict the topmost block on top of A: the topmost block B on A satisfies $above(\text{B}, \text{A}) \wedge clear(\text{B})$.

The same technique also applies to the problem of path-finding in any acyclic maze. Recall that a classical solution to any acyclic maze is that we only make right turns. This can be cast as a linearization of the entire maze by doing a depth-first search from the start or the goal location. In particular, starting from the goal location if we only do right turns, we will get a sequence of the edges of the maze such that the start edge is one of the elements (if there is a path from the start to the goal). Therefore, doing goal regression under this linearized maze is trivial: the entire "regression tree" is just a chain of edges and there is a simple shortcut for predicting the next move.

D.3 Analysis of AI Planning Problems

Here, we list six popular problems that have been widely used in AI planning communities and discuss their circuit depth and breadth. We summarize our results in the Table D.1.

Problem	Breadth	Regression Width	Depth
BlocksWorld	Constant	1	Unbounded
Logistics	Constant	0	Unbounded
Gripper	Constant	0	Constant
Rover	Constant	0	Unbounded
Elevator	Constant	0	Unbounded
Sokoban	Unbounded	N/A (not serializable)	Unbounded

Table D.1: **Width and circuit complexity analysis for 6 problems widely used in AI planning communities.** This table contains the same content as Table 4.1; we include it here for easy reference.

Note that here, we are limiting our discussion to the case where the goal of the planning problem is a single atom. In summary, most of these problems have a constant breadth (i.e., the regression width of the corresponding problem is constant) except for Sokoban. Most problems have an unbounded depth: that is, the depth of the circuit will depend on the number of objects (e.g., the size of the graph or the number of blocks. The problem gripper has a constant depth under a single-atom goal because it assumes the agent can move directly between any two rooms; therefore, no pathfinding is required. Sokoban has an unbounded breadth even for single-atom goals because if there are multiple boxes blocking the way to a designated position, the order to move the boxes can not be determined by simple rules. The constant breadth results can be proved by construction: list all regression rules in the domain. In the following Appendix D.3.1, we first discuss the SOS width of two representative problems, Blocks World and Logistics. All other problems can be analyzed in a similar way.

Before we delve into concrete complexity proofs, we would like to add the note that for problems in this list, when there are multiple goals, usually the goals are not serializable (in the optimal planning case). This can be possibly addressed by introducing "super predicates" that combine two literals. For example, if two goal atoms $p(x)$ and $q(x)$ are not serializable, we can introduce a new predicate $p_and_q(x)$, and rewrite all operators with this new super predicate. Appendix D.1.3 provides more discussion and examples. This will make the problem serializable but at the cost of significantly enlarging the set of predicates (exponentially with respect to the number of goal atoms).

If we only care about satisficing plans, for Logistics, Gripper, Rover, and Elevator, there

exists a simple serialization for any conjunction of goal atoms (basically achieving one goal at a time).

These theoretical analyses can be used in determining the size of the policy circuit needed for each problem. For example, one should use a RelNN of breadth $(k+1)\cdot\beta$, where k is the regression width and β is the max arity of predicates. For problems have unbounded depth, usually the depth of the circuit grows in $O(N)$, where N is the number of objects in the environment (e.g., in Elevator, the number of floors).

D.3.1 SOS Width Analysis

To prove that a problem can be solved optimally using S-GRS, we only need to show that there exists a goal regression rule selector. Therefore, we will be using the syntax from Section 4 to write down the set of goal regression rule selectors.

Finite SOS width for Blocks World.

$$\forall cons.\forall x.\forall y.on(y,x) \in s_0 \;\; clear(x)^{cons} \leftsquigarrow on(y,x), clear(y), handsfree() \parallel unstack(y,x)$$
$$\forall cons.\forall x.holding(x) \in s_0 \;\; clear(x)^{cons} \leftsquigarrow holding(x) \parallel place\text{-}table(x)$$
$$\forall cons.\forall x.\forall y.on(x,y) \in s_0 \;\; hoding(x)^{cons} \leftsquigarrow on(x,y), clear(x), handsfree() \parallel unstack(x,y)$$
$$\forall cons.\forall x.on\text{-}table(x) \in s_0 \;\; hoding(x)^{cons} \leftsquigarrow on\text{-}table(x), clear(x) \parallel pick\text{-}table(x)$$
$$\forall cons.\forall x. \;\; on\text{-}table(x)^{cons} \leftsquigarrow holding(x) \parallel place\text{-}table(x)$$
$$\forall cons.\forall x.\forall y. \;\; on(x,y)^{cons} \leftsquigarrow clear(y), holding(x) \parallel stack(x,y)$$
$$\forall cons.\forall x.holding(x) \in s_0 \;\; handsfree()^{cons} \leftsquigarrow holding(x) \parallel place\text{-}table(x)$$

Finite SOS width for Logistics.

$$\forall cons.\forall v.\forall o.\forall \ell. \;\; at(o,\ell)^{cons} \leftsquigarrow in(o,v), at(v,\ell) \parallel unload(y,x)$$
$$\forall cons.\forall v.\forall o.\forall \ell. \;\; in(o,v)^{cons} \leftsquigarrow at(o,\ell), at(v,\ell) \parallel load(y,x)$$
$$\forall cons.\forall v.\forall \ell_1.\forall \ell_2.\forall c. \;\; at(v,\ell_2)^{cons} \leftsquigarrow loc(\ell_1,c), loc(\ell_2,c), at(o,\ell_1) \parallel drive(v,\ell_1,\ell_2,c)$$
$$\forall cons.\forall v.\forall a_1.\forall a_2. \;\; at(v,a_2)^{cons} \leftsquigarrow at(v,a_1) \parallel fly(v,a_1,a_2)$$

Here, we have omitted static properties for type checking (such as a_1 and a_2 need to be airports).

D.4 Experimental Details

D.4.1 Training details

Environment setup. For the Assembly3 task, we first uniformly sample the set of type-A, type-B, and type-C object. Then, we randomly add *match* relationships so that there will be exactly only one tuple of (a, b, c) that satisfies the matching condition.

For the BlocksWorld-Clear task, we use the same random sampler for initial configurations as in Chapter 2. It iteratively adds blocks to an empty initial state. At each step, it randomly selects a clear object (or the table), and places a new object on top of it. After generating the initial state, we randomly sample an object that is not clear in the initial state as the target block.

For the Logistics task, we first randomly generate a tree rooted at the start node. This is done by iteratively attaching new nodes to a leaf node. Next, we randomly add a small number of edges to the tree. Finally, we randomly pick a node as the target node. Since we are only a small number of additional edges added to the tree, with high probability, the starting node and the end node are not in the same strongly connected component.

Additional hyperparameters. The breadth, width, and hidden dimension parameters for neural networks have been specified in the experiment section and they vary across tasks. For all experiments, we use the Adam optimizer to train the neural network, with a learning rate of $\epsilon = 0.001$. Additionally, we set $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

Compute. We trained with 1 NVIDIA Titan RTX per experiment for all datasets, from an internal cluster.

Code. We release the code for reproducing all our experimental results in https://github.com/concepts-ai/goal-regression-width.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix E

Appendix for Diffusion Continuous Constraint Satisfaction Solver

This chapter is organized as the following. Appendix E.1 describes how our datasets are collected for each task and how the inputs are encoded for diffusion models. We also provided discussions on sample efficiency of Diffusion-CCSP on the tasks studied. Appendix E.2 details hyper-parameter choices of our models, implementation details of StructDiffusion baseline [Liu *et al.*, 2023b], as well as training details for both. We have also included additional discussions on model training and fine-tuning, constraint weights, local optima in optimization, and partial observability. Appendix E.3 discusses common failure cases in solutions generated by out Diffusion-CCSP. Appendix E.4 includes a detailed experiment on the number of samples it takes for Diffusion-CCSP to solve CCSP problems in each task. Appendix E.5 discusses how to integrate Diffusion-CCSP for CCSP solving with task and motion planning (TAMP) algorithms. Appendix E.6 shows the learned energy landscapes for qualitative constraints in 2D shape arrangement tasks.

E.1 Problem Domains and Data Generation

In all datasets, the number the examples involving different number of objects are the same.

E.1.1 2D Triangle Packing

Data. The data is generated by randomly sampling points in the square and, connecting them with each other and the edge points using the Bowyer–Watson algorithm, then shrinking each triangle a little to make space among them.

Encoding. We define the resting pose of a triangle as the pose when the vertex A facing the shortest side BC is at the origin and its longest side AB is aligned along $+x$ axis. Its geometry is encoded using three numbers, the length $|AB|$ and the vector $(x, y) = AC$. Its pose is encoded as the 2D coordinates of vertex A, along with the *sin* and *cos* values of the rotation θ from the testing pose.

E.1.2 2D Shape Arrangement with Qualitative Constraints

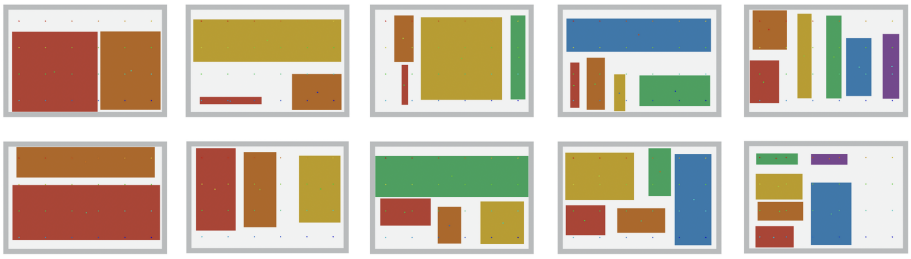


Figure E-2: **Example rearrangements of rectangles with 11 types of qualitative constraints.**

Data. The data is generated by recursively splitting the tray at different proportions until depth 3. For each resulting region, a random padding is added to each side. Regions whose area or side is too small are discarded. Labels of qualitative constraints are created by hand-crafted rules, e.g. `'close-to'` means the distance between two objects is smaller than the maximum width of two objects. All qualitative constraints include `'center-in'`, `'left-in'`, `'right-in'`, `'top-in'`, `'bottom-in'`, `'left-of'`, `'top-of'`, `'close-to'`, `'away-from'`, `'h-aligned'`, and `'v-aligned'`.

Encoding. The resting pose of a rectangle is when its longer side is oriented horizontally. Its geometry is encoded using its width and length at resting pose. Its pose is the 2D pose of the centroid, along with the *sin* and *cos* encoding of the object's *yaw* rotation. Many problems require some objects to be at vertical positions in order for all the constraints to be satisfied.

E.1.3 3D Object Stacking with Stability Constraints

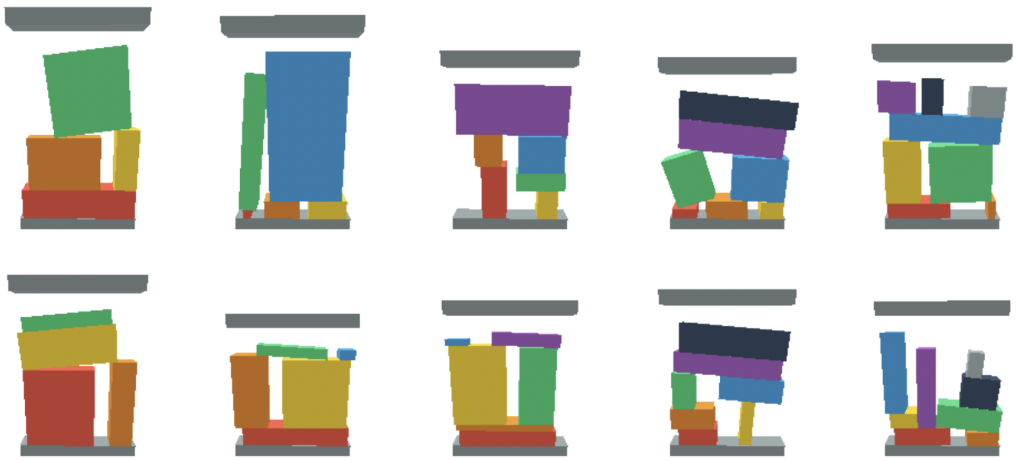


Figure E-3: **Example stable configurations of rectangles**, with `in`, `cfree`, and `supported-by` constraints. There is at least one object that's supported by multiple objects.

Data. The data is generated by randomly splitting a 2D vertical region, then shrinking and cutting each region. The resulting cuboids are initiated in PyBullet simulator and letting them drop until rest. We filter for configurations where the final state is stable. Then all objects are removed one by one to test that each intermediate configuration is also stable. An upper shelf is added to be just enough to fit the current configuration with a little overhead space.

Encoding. The resting pose of a cuboid is when its longer side is oriented horizontally. The cuboid's geometry is encoded using its width and length. The cuboid's pose is the 2D pose of the centroid, along with the *sin* and *cos* encoding of the cuboid's *roll* rotation.

E.1.4 3D Robot Packing with Robots



Figure E-4: **Example configuration of 3D shapes** that enables the robot place each object in a given sequence without colliding into other objects already placed. All the bottles, dispensers, and bowls affords only side grasps.

Data. The 10 categories of objects used in this dataset include `'Dispenser'`, `'Bowl'`, `'StaplerFlat'`, `'Eyeglasses'`, `'Pliers'`, `'Scissors'`, `'Camera'`, `'Bottle'`, `'BottleOpened'`, and `'Mug'`, contributing in total 53 object assets. For each asset, a fixed number of grasps are generated beforehand that points directly to one of its five faces, e.g. $+x, -x, +y, -y, +z$. The problems are generated by randomly splitting the tray into rectangle regions, then fitting into each region an object with random category, asset, and scale. An order to place the objects is sampled, then a TAMP planner is called to check whether the corresponding grasps can be found (by enumerating all combinations of order and grasps) so that the gripper at each grasp pose won't collide with any objects already in the goal region and all arm trajectories going to the grasp configurations can be found.

Encoding. The objects' geometry is encoded using bounding box (e.g., width, length, height). An object's grasp is encoded using a five-dimensional vector, indicating the face that the gripper points to. For example, $[0, 0, 0, 0, 1]$ is a top grasp. The object's pose is encoded using its 3D coordinate in the tray frame, along with the *sin* and *cos* encoding of the object's *yaw* rotation.

E.1.5 Discussion: Sample Complexity of Diffusion-CCSP

Our training datasets are on the scale of 10k-30k for each task. It is always important to be aware of the amount of training data needed and the cost of obtaining it from various sources. We observed that we can train the models in simulation with abundant data and believe they can be applied to real-world robotic tasks. In our system, we learn two categories of constraints. First, geometric and physical constraints are independent both of the robot

(and hence of perceptual and motor noise and irregularities that cause most sim-to-real difficulties) and of any human input. Learning to check that object placements are collision free, or that robot configurations are kinematically feasible can be done entirely via data obtained in simulation and transfer without diffficulty to the real world. Second, qualitative constraints corresponding to human input must be obtained from human annotation. Such annotations are costly obtain and, as a matter of future work, we would like to leverage additional data sources, such as object relationship information in large vision/language models.

E.1.6 Discussion: Partial Observability

Across all experiments in the paper, we have been assuming a fully-observed setting where we know object shapes (in particular, their bounding boxes) and their initial 3D poses. We think one possible way to handle at least some types of partial observability in our setting is through replanning. Taking object packing as a concrete example, given the segmented (partial) point clouds for objects, we use learning or heuristic methods to complete the object mesh and then run our solver to compute plans. During execution, assuming we can sense object collisions while moving them, we can always replan for a different placement in a closed-loop manner.

E.2 Implementation and Training Details

We used 1000 diffusion timesteps for all models, except for Diffusion-CCSP on Task 4 robot packing, which used 500 diffusion timesteps. In neural network input encoding, all object dimensions and poses are normalized with regard to that of the container or shelf region.

Diffusion-CCSP: Given the geometry, pose, and grasp features, we use geometry, pose, and grasp encoders to process them to the same hidden dimension of 256, which is the same dimension as time embedding. The outputs from individual diffusion models have the same dimension as input pose embedding. The output for each object's pose are composed by taking the average of all outputs of diffusion models that have the object as an argument. They are processed through a pose decoder to get the output reconstructed noise value added to the poses. The encoders, pose decoder, and diffusion models are trained together using the L2 loss between reconstructed noise value and input noise value, as detailed in Section 5.2.2. For our ULA variant, we used sampling steps of 10 and step sizes of $2\beta(t)$ where β is cosine beta schedule for the diffusion model.

Rejection-Sampling: The objects are sampled in sequence inside the container (collision-free with the container is guaranteed). For each object, its pose is sampled uniformly at

random at most 50 times to find a placement that's collision-free to all objects already placed.

StructDiffusion: We implemented the architecture of concatenating the geometry and pose embeddings for each object as one token (dimension 512 for each token) to the transformer encoder, plus positional encoding and time encoding. For task 4, we also concatenated the chosen grasp embedding to the object embedding (total dimension 768 for each token). We used four layers of residual attention blocks and two heads. We added normalization layers before and after the transformer layers. We used the same diffusion timesteps as our Diffusion-CCSP models. Because there is no language input, we omitted the word embeddings from the architecture.

E.2.1 Discussion: Individual Constraint Diffusion Training and Finetuning

Our diffusion models for individual constraints can be trained either jointly or independently; however they are trained, they can be applied in novel combinations at performance time. In our experiments, we concurrently train all constraints since our data naturally encompasses multiple constraints in a single scene, such as object collision-free constraints in a bin packing problem. It is also possible to finetune a pretrained constraint solver with new data. In some scenarios such as generalization to unseen object shapes this might be very helpful.

E.2.2 Discussion: Weights for Energy Function Composition

In general, selecting appropriate weights for different constraints is important and challenging, which has been observed in Urain *et al.* [2023] and Liu *et al.* [2022a]. In our experiments, we consistently use weight 1 for all constraints, and this setting has proven effective in our tasks. We postulate two reasons for this. First, visualizations of the learned energy functions (see Appendix E.6) indicate that the energy field is steep. The areas in the configuration space that meet the constraints exhibit considerably lower energy than areas that don't. As such, when constraints are combined, the solver tends to prioritize solutions that satisfy all constraints. Second, during training, our models are jointly trained to explain data with various combined constraints. This implicitly forces the diffusion models to learn score functions that support combinations.

E.2.3 Discussion: Local Optima in Optimization

Bad local optima can be an issue for multi-constraint optimization problems. In this paper, we partially address this problem by running multiple seeds in parallel. That is, given a

CCSP problem, we run the reverse diffusion process with multiple randomly sampled initial noise values. For instance, in the 3D object stacking task, on average, 33.3 samples are needed for solving tasks involving 5 objects. This does not significantly affect our runtime because different random seeds can be run in parallel. In two cases shown in the table (object packing and stable stacking), we have used the physical simulator to check solution validity—whether objects are in collision-free poses and whether the system is physically stable. In other cases where solution checkers are unavailable, we can consider learning classifiers for conditions.

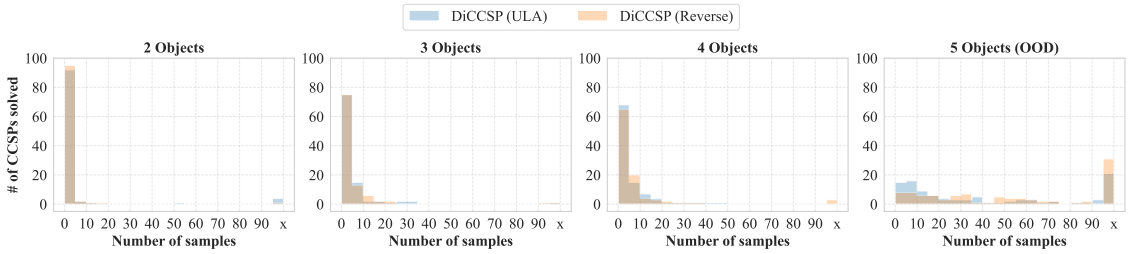
E.3 Failure Case Analysis

Example failure cases for Task 2 involving qualitative constraints are shown in Figure E-5. One insight we gained is that *the failure rate to satisfy a qualitative constraint is inversely correlated with the area of its defining region*. We ran a Diffusion-CCSP 4000 times to solve a test set of 400 CCSPs involving 2-4 shapes and counted the number of unsatisfied constraints in solutions generated by Diffusion-CCSP. From Figure E-6a, we make two observations: (1) the failure rate is inversely correlated with the amount of data for each constraint, and (2) constraints with discontinuous regions are harder to satisfy (e.g. v-aligned, close-to). Then, we look at a problem with one object A (size 0.5 by 0.5) in the center of a container. We estimate the area of the defining region of each constraint by doing rejection sampling on the pose (x, y) of object B (size 0.5 by 0.5) that doesn't collide with A (except for *center-in*). From Figure E-6b, the order is highly similar to the order in Figure E-6a, supporting the hypothesis that the smaller the defining region of a constraint, the harder it is to satisfy. Note that this experiment assumes two object. Therefore, although the area is large for *cfree* in this estimate, it doesn't represent the dataset where there are 2 - 5 shapes in each problem. So we exclude it from further discussions.

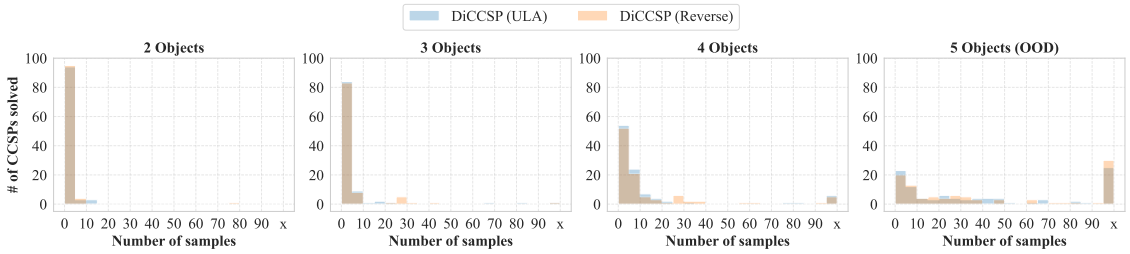
Example failure cases for Task 1 and 3 are shown in Figure E-7 and Figure E-8.

E.4 Additional Sampler Comparison

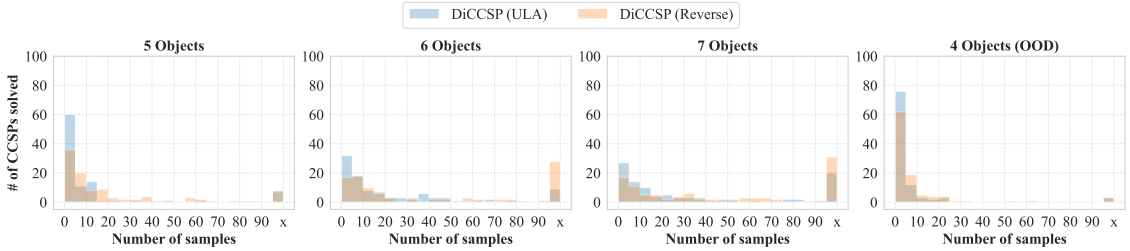
In the main experiment section, we let Diffusion-CCSP and baselines generate 10 samples for each CCSP and check if all constraints are satisfied. Here we let Diffusion-CCSP generate 100 samples on 100 problems, and plot the number of problems it took Diffusion-CCSP to solve for each task. In a batch of 100 CCSPs, it takes on average 0.01-0.04 sec for Diffusion-CCSP (Reverse) to solve each CCSP while 0.06-0.19 sec for Diffusion-CCSP (ULA) to solve each CCSP, as shown in Table E.1.



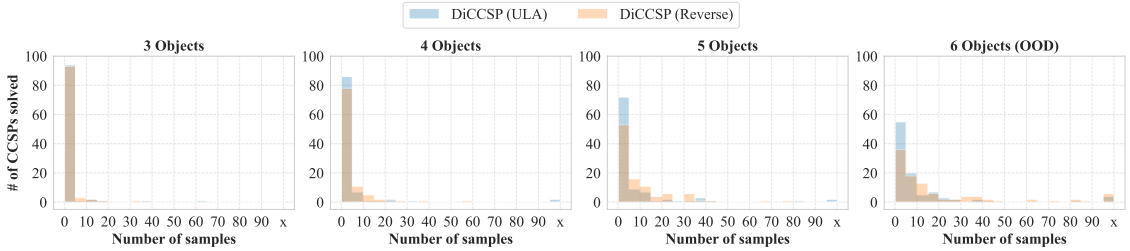
(a) Task 1: 2D Triangle Packing



(b) Task 2: 2D Shape Arrangement with Qualitative Constraints



(c) Task 3: 3D Object Stacking with Stability Constraints



(d) Task 4: 3D Object Packing with Robots

Figure E-9: **Number of Diffusion-CCSP runs it takes to solve 100 CCSPs.** OOD means problems that are out of training distribution (i.e. include more objects than trained on). On the x-axis, tick `x` means problems not solved within 100 samples.

E.5 Integration with Task and Motion Planning Algorithms

So far, we have presented a generic solution to solving constraint satisfaction problems that involve geometric and physical constraints. However, assuming that the constraint graph is given as input to the algorithm. This approach can be directly used to solve particular



Figure E-1: **Example collision-free configurations of triangles.**

	2 Objects	3 Objects	4 Objects	5 Objects
Diffusion-CCSP (ULA)	0.10	0.10	0.11	0.12
Diffusion-CCSP (Reverse)	0.01	0.01	0.01	0.01
StructDiffusion	0.08	0.08	0.08	0.08

(a) Task 1. 2D Triangle Packing

	2 Objects	3 Objects	4 Objects	5 Objects
Diffusion-CCSP (ULA)	0.17	0.18	0.19	0.19
Diffusion-CCSP (Reverse)	0.04	0.04	0.04	0.04

(b) Task 2. 2D Shape Arrangement with Qualitative Constraints

	4 Objects	5 Objects	6 Objects	7 Objects
Diffusion-CCSP (ULA)	0.15	0.16	0.17	0.19
Diffusion-CCSP (Reverse)	0.01	0.01	0.01	0.02

(c) Task 3: 3D Object Stacking with Stability Constraints

	3 Objects	4 Objects	5 Objects	6 Objects
Diffusion-CCSP (ULA)	0.06	0.06	0.07	0.07
Diffusion-CCSP (Reverse)	0.01	0.01	0.01	0.01
StructDiffusion	0.09	0.09	0.08	0.08

(d) Task 4: 3D Object Packing with Robots

Table E.1: **Average runtime (seconds) of models in a batch of 100 CCSPs.**

tasks such as the pose prediction task in object rearrangements. Next, we illustrate how the proposed method can be integrated with a search algorithm to solve general task and motion planning (TAMP) problems, where the constraint graphs are automatically constructed based on the sequence of actions that has been applied and the goal specification of the task. For brevity, we will present a simplified formulation of TAMP problems. For more details, please refer to the recent survey [Garrett *et al.*, 2021].

Formally, given a space $\mathcal{S}$ of world states, a problem is a tuple $\langle \mathcal{S}, s_0, \mathcal{G}, \mathcal{A}, \mathcal{T} \rangle$, where $s_0 \in \mathcal{S}$ is the initial state (e.g., the geometry and poses of all objects), $\mathcal{G} \subseteq \mathcal{S}$ is a goal specification (e.g., as a logical expression that can be evaluated based on a state: $in(\text{A}, \text{Box})$ *and* $in(\text{B}, \text{Box})$), $\mathcal{A}$ is a set of continuously parameterized actions that the agent can execute (e.g., pick-and-place), and $\mathcal{T}$ is a partial environmental transition model $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. Each action a is parameterized by two functions: precondition pre_a and effect eff_a. The semantics of this parameterization is that: $\forall s \in \mathcal{S}. \forall a \in \mathcal{A}. pre_a(s) \Rightarrow (\mathcal{T}(s, a) = eff_a(s))$.

The goal of task and motion planning is to output a sequence of actions $\bar{a}$ so that the terminal state s_T induced by applying a_i sequentially following $\mathcal{T}$ satisfies $s_T \in \mathcal{G}$. The state space and action space are usually represented as STRIPS-like representations: the state is a collection of state variables and each action is parameterized by a set of arguments. Figure E-10 shows a simplified definition of the pick-and-place action in a table-top manipulation domain.

A characteristic feature of TAMP problems is that the decision variables (i.e., the arguments of actions) include both discrete variables (e.g., object names) and continuous variables (e.g., poses and trajectories). Therefore, one commonly used solution is to do a bi-level search [Dantam *et al.*, 2018; Garrett *et al.*, 2020a]: the algorithm first finds a plan that involves only discrete objects (called a *plan skeleton*) and uses a subsequent CSP solver to find assignments of continuous variables, backtracking to try a different high-level plan if the CCSP is found to be infeasible. For example, consider the discrete task plan (also called *plan skeleton*): *pick-and-place*(A), *pick-and-place*(B). There are six parameters to decide: the grasps on A and B, the place locations of A and B, and the robot trajectory while transporting A and B. Based on the preconditions of actions and the goal condition of the task, we can obtain the constraint graph. Therefore, each solution to the CCSP constructed based on the plan skeleton corresponds to a concrete plan of the original planning problem. By combining the high-level search of plan skeletons and our constraint solver Diffusion-CCSP, the integrated algorithm is capable of solving a wide range of task and motion planning problems that involves geometric, physical, and qualitative constraints.

E.6 Learned Gradient Fields

We visualized the learned gradient functions for all qualitative constraints trained for Task 2 (Figure E-12 column 3). The gradient functions for the first 6 constraints (from `in` to `bottom-in`) corresponds to the direction of changing the pose of a square shape of size 0.15 by 0.15 inside a container of size 1 by 1 given different initial poses. The gradient functions for the later 7 constraints (from `cfree` to `v-aligned` in Figure E-13) corresponds to the direction of updating the pose of a square shape of size 0.15 by 0.15 inside a container of size 1 by 1 where another square shape of the same size has been fixed at pose $(0, 0)$. We show a pose sampled by Diffusion-CCSP for the target object in column 4. In column 5, we visualized the distribution of poses that satisfy each qualitative constraint in the training data. Note that the poses are normalized against the size of the container and the shapes of those objects are no longer 0.15 by 0.15.

We also computed the energy function as the difference between input noise and reconstructed output at the final step of diffusion (Figure E-12 column 2). Note that the energy field is steep. Therefore the areas in the configuration space that meet the constraints exhibit considerably lower energy than areas that don't. As such, when constraints are combined, the solver tends to prioritize solutions that satisfy all constraints.

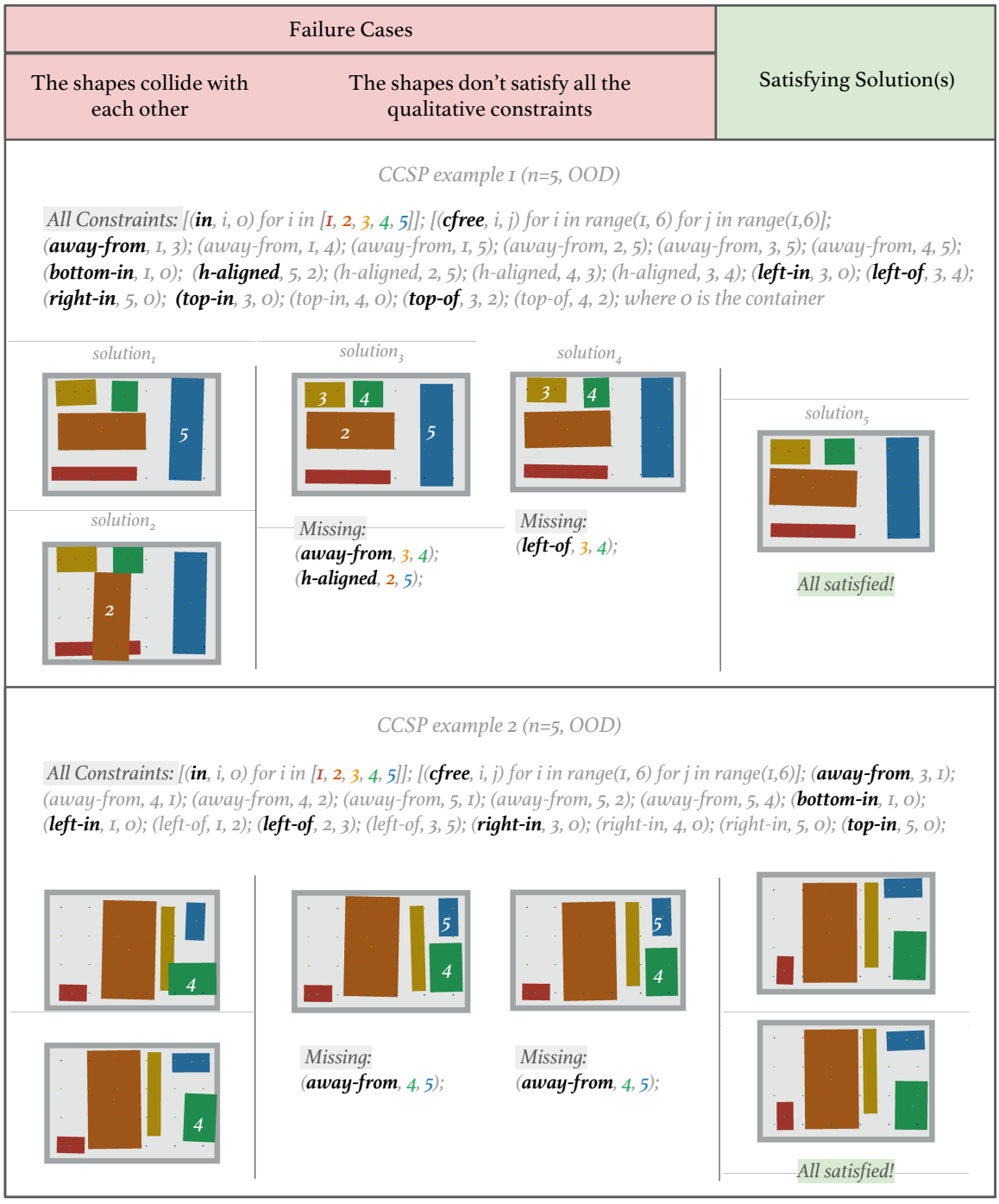
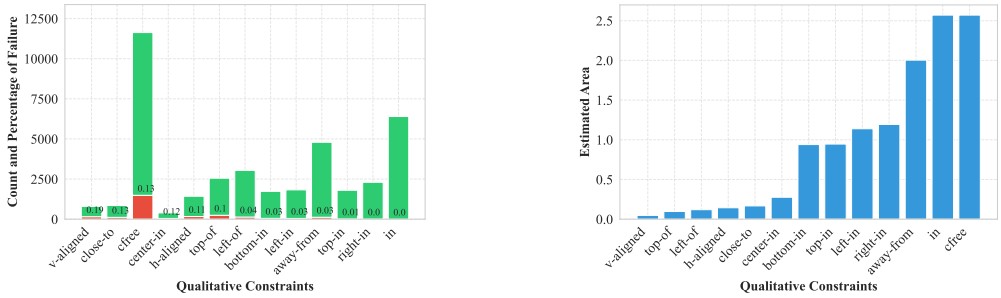
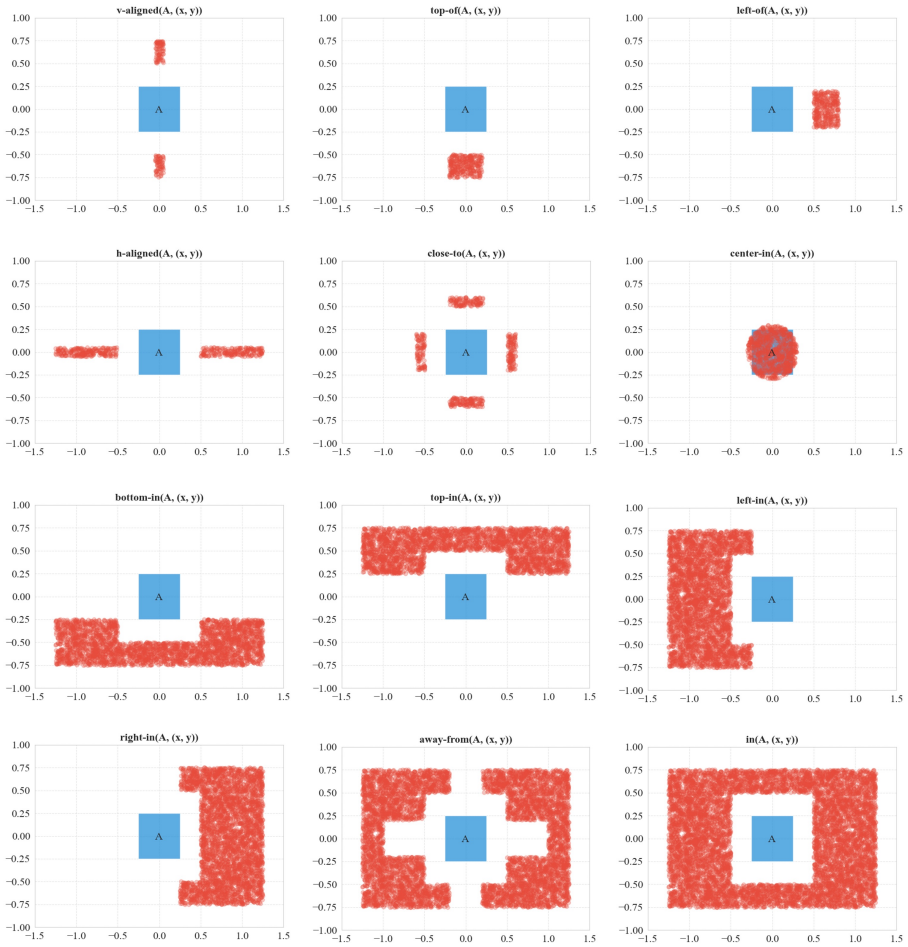


Figure E-5: **Failure and success samples for Task 2: 2D shape rearrangement with qualitative constraints.** All solutions are generated by Diffusion-CCSP (ULA), trained on 30k data up to 4 objects.



(a) **Failure rate of qualitative constraints**. The failure rate of each constraint is annotated on the bars. The red bars denote the number of failures, while the green bars denote success. Constraints are sorted by failure rate.

(b) **Estimated area occupied by qualitative constraints.** The order is highly similar to the order in (a), supporting the hypothesis that the smaller the defining region of a constraint, the harder it is to satisfy.



(c) **Training data distribution for each constraint.** Examples are ordered by increasing area of the defining region. Each point is the centroid for an object A that appeared in one *constraint*(A, B) during training.

Figure E-6: **Analysis of how the failure rate of qualitative constraints correlates with the area of its defining region.**

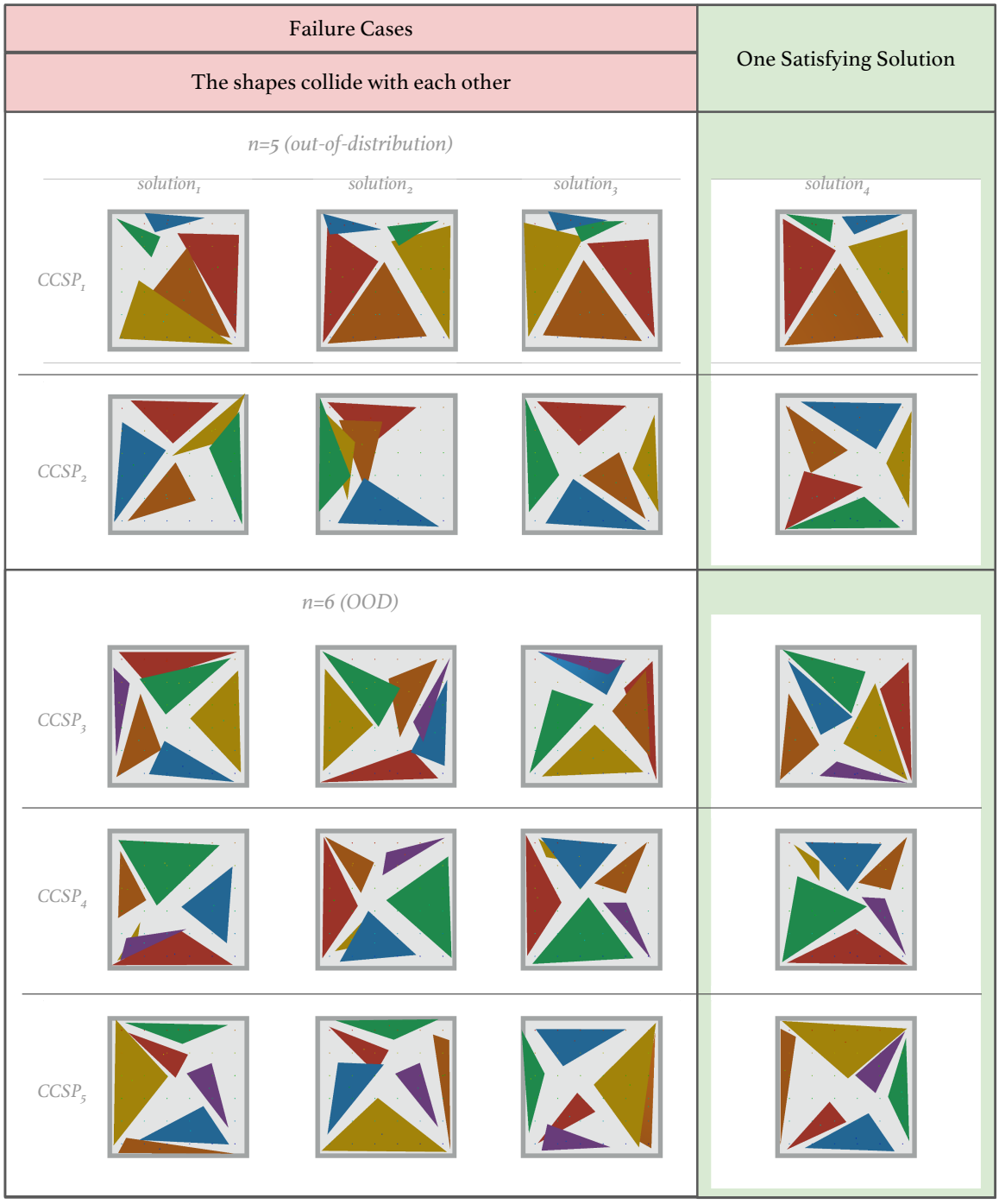


Figure E-7: **Failure and success samples for Task 1: 2D triangle packing.** All solutions are generated by Diffusion-CCSP (ULA), trained on 30k data up to 4 objects.

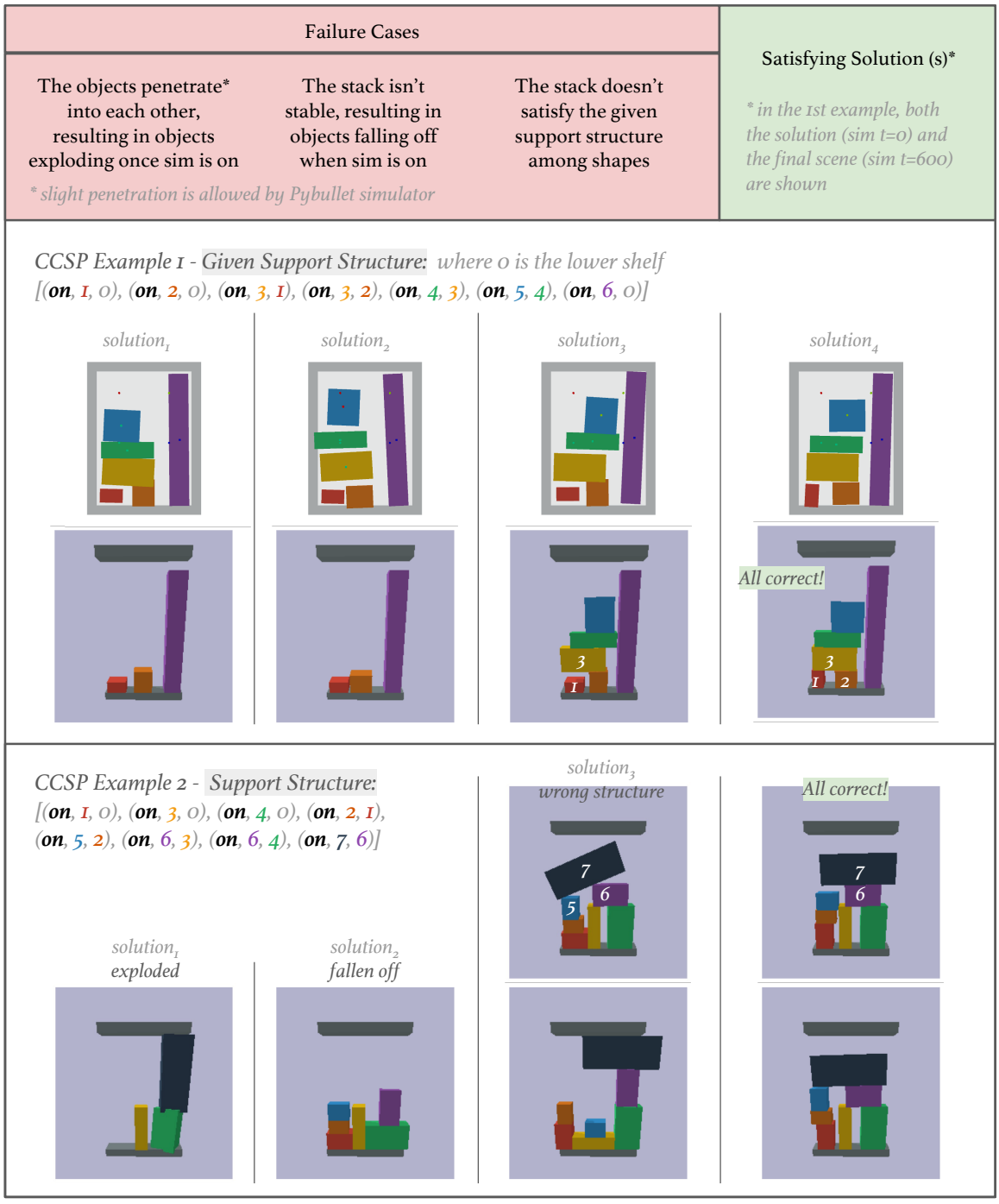
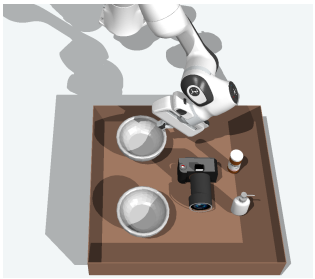


Figure E-8: **Failure and success samples for Task 3: 3D object stacking with stability constraints.** All solutions are generated by Diffusion-CCSP (ULA), trained on 24k data up to 7 objects.



```
;; pick up x and and place x to p.
action pick-place(x, g, p, t)
  pre: valid-grasp(x, pose[x], g)        ;; g is the grasp pose on x
       valid-traj(x, pose[x], g, p, t)   ;; p is the target pose of x
       forall z. cfree(x, z, pose[x], t) ;; t is the robot trajectory
  eff: pose[x] := p                      ;; object x will be moved

goal: in(A, C, pose[A], pose[C]) and in(B, C, pose[B], pose[C])
```

The precondition-effect definition of a continuously parameterized action pick-place: pick and place object x.

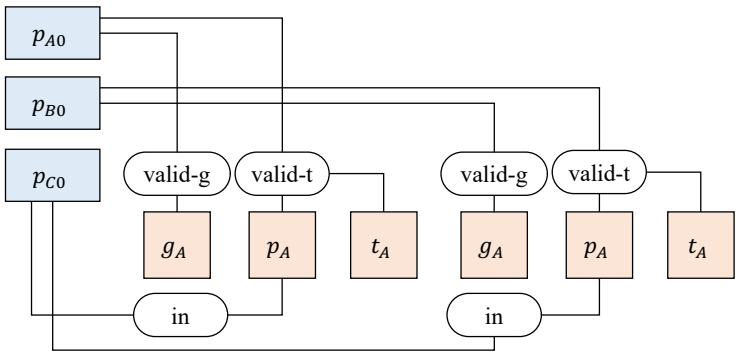
Figure E-10: **Illustration of a simple task and motion planning problem.** The domain contains only one action: pick-and-place of objects. It contains three preconditions: g should be a valid grasp pose on object x, t should be a valid robot trajectory that moves x from its current pose to the target pose p, and during the movement, the robot and the object x should not collide with any other objects y. If successful, the object will be moved to the new location. The goal of the task is to pack both objects into the target box without any collisions.

```
pick-place(A, gA, pA, tA)
 valid-grasp(A, pA0, gA)
 valid-traj(A, pA0, gA, pA, tA)
 cfree(A, B, pB0, tA)
 cfree(A, C, pC0, tA)

pick-place(B, gB, pB, tB)
 valid-grasp(B, pB0, gB)
 valid-traj(B, pB0, gB, pB, tB)
 cfree(B, A, pA, tB)
 cfree(B, C, pC0, tB)

Goal
 in(A, C, pA, pC0)
 in(B, C, pB, pC0)
```

(a) A partially specified plan skeleton and its corresponding constraints.



(b) The corresponding TAMP constraint graph derived from the plan skeleton. cfree constraints and geometric features are omitted for brevity.

Figure E-11: **Example of a partially specified plan skeleton in task and motion planning, together with the corresponding set of constraints.** p_{A0}, p_{B0}, and p_{C0} corresponds to the initial pose of objects.

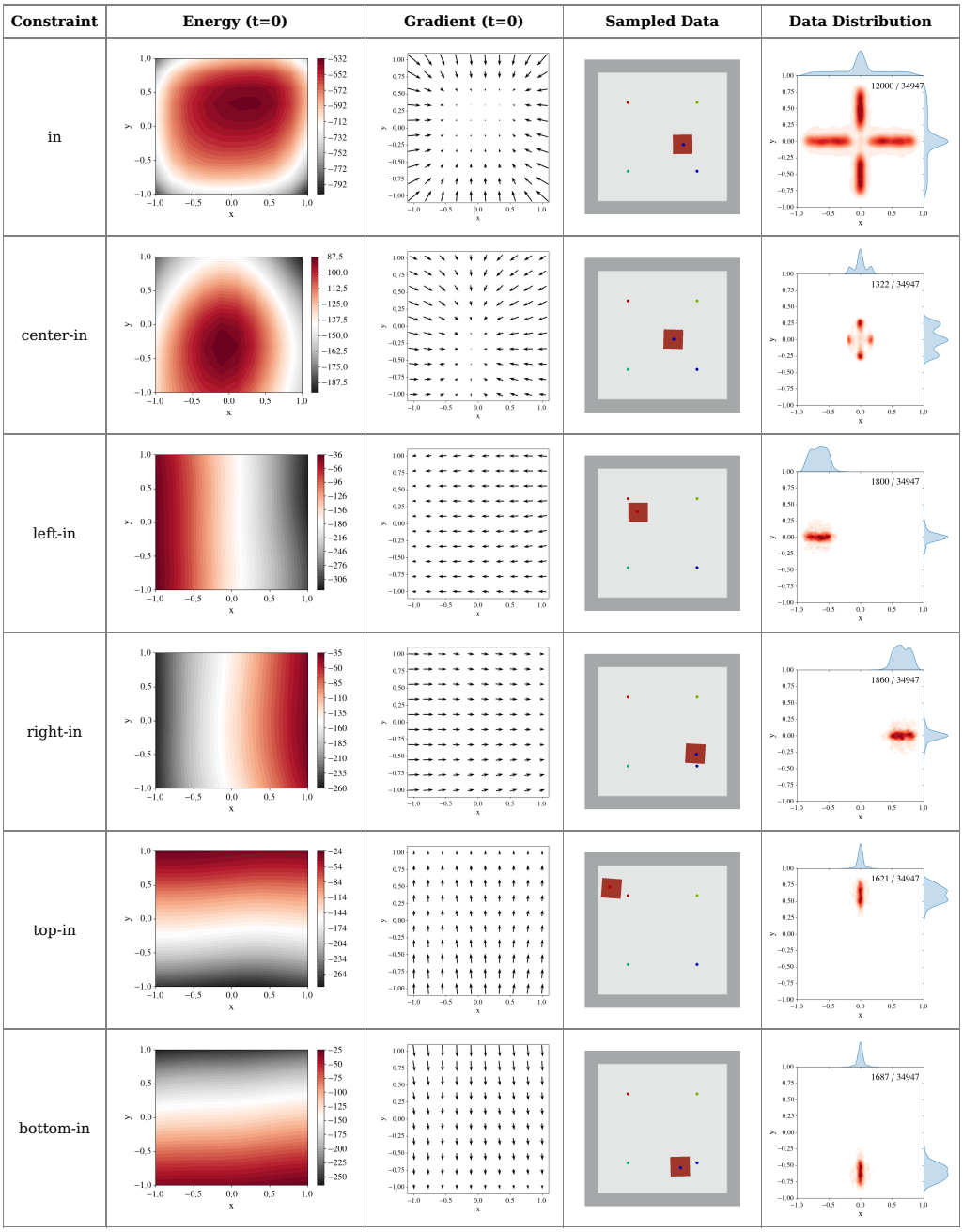


Figure E-12: **Visualization of energy and gradient functions of trained diffusion models for qualitative constraints (Part 1. constraints between a shape and the container). Column 2**: Energy fields inside a container of size 1 by 1. **Column 3**: Gradient functions that update poses in different time steps. **Column 4**: Poses sampled according to the given qualitative constraint. **Column 5**: Poses in the training data set with the given constraint, for 30,000 CCSPs that involve 2 - 4 objects that have different sizes.

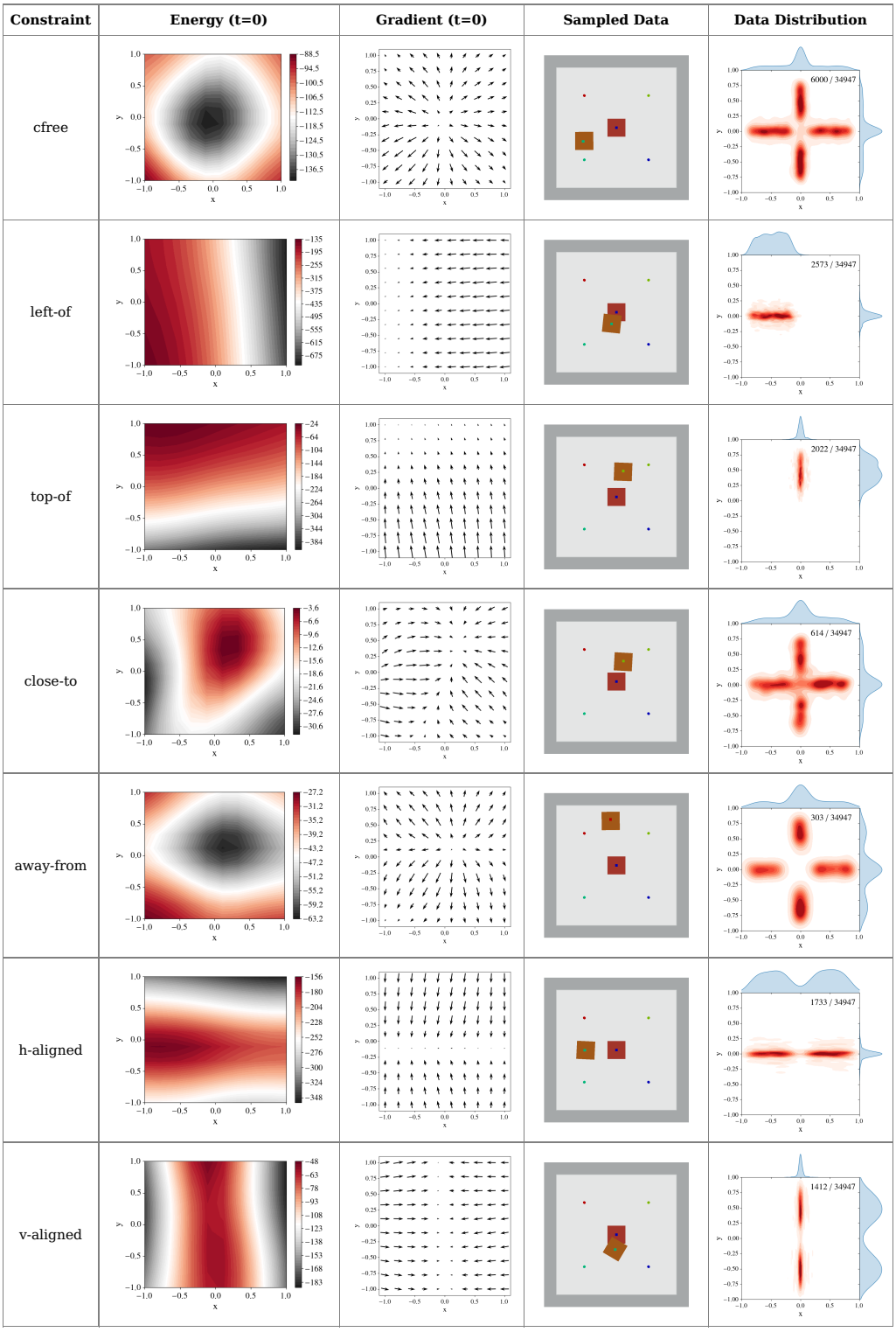


Figure E-13: **Visualization of energy and gradient functions for qualitative constraints (Part 2. constraints between two shapes).**

Appendix F

Appendix for Visual Concept Learning

F.1 CLEVR Domain-Specific Language

We first introduce the domain-specific language (DSL) designed for the CLEVR VQA dataset [Johnson *et al.*, 2017a]. Table F.1 shows the available operations in the DSL, while Table F.2 explains the type system.

We note that some function takes Object as its input instead of ObjectSet. These functions require the uniqueness of the referral object. For example, to answer the question "What's the color of the red object?", there should be one and only one red object in the scene. During the program execution, the input object set will be implicitly cast to the single object (if the set is non-empty and there is only one object in the set). Such casting is named `Unique` in related works [Johnson *et al.*, 2017b].

F.2 Semantic Parsing

As shown in Appendix F.1, a program can be viewed as a hierarchy of operations which take concepts as their parameters. Thus, NS-CL generates the hierarchies of latent programs in a sequence to tree manner [Dong and Lapata, 2016]. The semantic parser adopts an encoder-decoder architecture, which contains four neural modules: (1) a bidirectional GRU encoder `IEncoder` [Cho *et al.*, 2014] to encode an input question into a fixed-length embedding, (2) an operation decoder `OpDecoder` that determines the operation tokens, such as `Filter`, in the program based on the sentence embedding, (3) a concept decoder `ConceptDecoder` that selects concepts appeared in the input question as the parameters for certain operations (e.g., `Filter` takes an object-level concept parameter while `Query` takes an attribute), and (4) a set of output encoders $\{\texttt{OEncoder}_i\}$ which encode the decoded operations by `OpDecoder` and output the latent embedding for decoding the next operation. The operation decoder,

the concept decoder, and the output encoders work jointly and recursively to generate the hierarchical program layout. Algorithm 12 illustrates the algorithmic outline of the semantic parser.

Algorithm 12 The String-to-Tree Semantic Parser.

function PARSE(f, $\{c_i\}$)
  program ← `EmptyProgram`()
  program.op ← `OpDecoder`(f)
  if *program.op* requires a concept parameter **then**
    program.concept ← `ConceptDecoder`(f, $\{c_i\}$)
  for $i = 0, 1, \cdots$ number of non-concept inputs of *program.op* **do**
    program.input[i] ← parse($\texttt{OEncoder}_i(f$, *program.op*) ,$\{c_i\}$)
  return *program*

The function *parse* takes two inputs: the current decoding state f and all concepts appeared in the question, as a set $\{c_i\}$. The parsing procedure begins with encoding the input question by `IEncoder` as f_0, extracting the concept set $\{c_i\}$ from the input question, and invoking $parse(f_0, \{c_i\})$.

The concept set $\{c_i\}$ is extracted using hand-coded rules. We assume that each concept (including object-level concepts, relational concepts, and attributes) is associated with a single word in the question. For example, the word "red" is associated with the object-level concept `Red`, while the word "shape" is associated with the attribute `Shape`. Informally, we call these words *concept words*. For a given question Q, the corresponding concept set $\{c_i\}$ is composed of all occurrences of the *concept words* in Q. The set of *concept words* is known for the CLEVR dataset. For natural language questions, one could run POS tagging to find all *concept words* [Andreas *et al.*, 2016a; Schuster *et al.*, 2015]. We leave the automatic discovery of concept words as a future work [Gauthier *et al.*, 2018]. We use the word embedding of the *concept words* as the representation for the concepts $\{c_i\}$. Note that, these "concept embeddings" are only for the program parsing. The visual module has separate concept embeddings for aligning object features with concepts in the visual-semantic space.

We now delve into the main function $parse(f, \{c_i\})$: we first decode the root operation *op* of the hierarchy by `OpDecoder`(f). If *op* requires a concept parameter (an object-level concept, a relational concept, or an attribute), `ConceptDecoder` will be invoked to choose a concept from all concepts $\{c_i\}$. Assuming *op* takes two non-concept inputs (e.g., the operation `Intersection` takes two object sets as its input), there will be two branches for this root node. Thus, two output encoders $\texttt{OEncoder}_0$ and $\texttt{OEncoder}_1$ will be applied to transform the current state f into two sub-states f_1 and f_2. *parse* will be recursively invoked based on f_1 and f_2 to generate the two branches respectively. In the DSL, the number of

non-concept inputs for any operation is at most 2.

In our implementation, the input encoder `IEncoder` first maps each word in the question into an embedding space. The word embeddings are composed of two parts: a randomly initialized word embedding of dimension 256 and a positional embedding of dimension 128 [Gehring *et al.*, 2017]. For a *concept word*, its word embedding only depends on which type it belongs to (i.e. object-level, relational or attribute). Thus, after being trained on a fixed dataset, the semantic parser can parse questions with novel (unseen) *concept words*. The sequence of word embeddings is then encoded by a two-layer GRU with a hidden dimension of $256 * 2$ (bidirectional). The function *parse* starts from the last hidden state of the GRU, and works recursively to generate the hierarchical program layout. Both `OpDecoder` and `ConceptDecoder` are feed-forward networks. `ConceptDecoder` performs attentions over the representations of all concepts $\{c_i\}$ to select the concepts. Output encoders $\texttt{OEncoder}_0$ and $\texttt{OEncoder}_1$ are implemented as GRU cells.

Another pre-processing of the sentence is to group consecutive object-level *concept words* into a group and treat them together as a single concept, inspired by the notion of "noun phrases" in natural languages. The computational intuition behind this grouping is that, the latent programs of CLEVR questions usually contain multiple consecutive `Filter` tokens. During the program parsing and execution, we aim to fuse all such `Filter`s into a single `Filter` operation that takes multiple concepts as its parameter.

A Running Example As a running example, consider again the question "What is the color of the cube right of the red matte object?". We first process the sentence (by rules) as: "What is the <Attribute 1 (color)> of the <(ObjConcept 1 (cube)> <RelConcept 1 (right)> of the <ObjConcept 2 (red matte object)>?". The expected parsing result of this sentence is:

```
Query(<Attribute 1>,
      Filter(<ObjConcept 1>,
             Relate(<RelConcept 1>,
                    Filter(<ObjConcept 2>, Scene)
             )
      )
).
```

The semantic parser encode the word embeddings with `IEncoder`. The last hidden state of the GRU will be used as f_0. The word embeddings of the *concept words* form the set $\{c_i\}$ = {Attribute 1, ObjConcept 1, RelConcept 1, ObjConcept 2}. The function *parse* is then invoked recursively to generate the hierarchical program layout. Table F.3 illustrates the decoding process step-by-step.

F.3 Program Execution

In this section, we present the implementation of all operations listed in Table F.1. We start from the implementation of Object-typed and ObjectSet-typed variables. Next, we discuss how to classify objects by object-level concepts or relational concept, followed by the implementation details of all operations.

Object-typed and ObjectSet-typed variables. We consider a scene with n objects. An Object-typed variable can be represented as a vector Object of length n, where $\text{Object}_i \in [0, 1]$ and $\sum_i \text{Object}_i = 1$. Object_i can be interpreted as the probability that the i-th object of the scene is being referred to. Similarly, an ObjectSet-typed variable can be represented as a vector ObjectSet of length n, where $\text{ObjectSet}_i \in [0, 1]$. ObjectSet_i can be interpreted as the probability that the i-the object is in the set. To cast an ObjectSet-typed variable ObjectSet as an Object-typed variable Object (i.e., the `Unique` operation), we compute: $\text{Object} = \text{softmax}(\sigma^{-1}(\text{ObjectSet}))$, where $\sigma^{-1}(x) = \log(x/(1-x))$ is the logit function.

Concept quantization. Denote o_i as the visual representation of the i-th object, OC the set of all object-level concepts, and A the set of all object-level attributes. Each object-level concept oc (e.g., `Red`) is associated with a vector embedding v^{oc} and a L1-normalized vector b^{oc} of length $|A|$. b^{oc} represents which attribute does this object-level concept belong to (e.g., the concept `Red` belongs to the attribute `Color`). All attributes $a \in A$ are implemented as neural operators, denoted as u^a (e.g., $u^{\texttt{Color}}$). To classify the objects as being `Red` or not, we compute:

$$\Pr[\text{object } i \text{ is } \texttt{Red}] = \sigma\left(\sum_{a \in A}\left(b_a^{\texttt{Red}} \cdot \frac{\langle u^a(o_i), v_{\texttt{Red}}\rangle - \gamma}{\tau}\right)\right),$$

where σ denotes the Sigmoid function, $\langle \cdot, \cdot \rangle$ the cosine distance between two vectors. γ and τ are scalar constants for scaling and shifting the values of similarities. By applying this classifier on all objects we will obtain a vector of length n, denoted as ObjClassify(`Red`). Similarly, such classification can be done for relational concepts such as `Left`. This will result in an $n \times n$ matrix RelClassify(`Left`), where $\text{RelClassify}(\texttt{Left})_{j,i}$ is the probability that the object i is left of the object j.

To classify whether two objects have the same attribute (e.g., have the same `Color`), we compute:

$$\Pr[\text{object } i \text{ has the same } \texttt{Color} \text{ as object } j] = \sigma\left(\frac{\langle u^{\texttt{Color}}(o_i), u^{\texttt{Color}}(o_j)\rangle - \gamma}{\tau}\right),$$

We can obtain a matrix AEClassify(`Color`) by applying this classifier on all pairs of objects,

where AEClassifier(`Color`)$_{j,i}$ is the probability that the object i and j have the same `Color`.

Quasi-symbolic program execution. Finally, Table F.4 summarizes the implementation of all operators. In practice, all probabilities are stored in the log space for better numeric stability.

F.4 Optimization of the Semantic Parser

To tackle the optimization in a non-smooth program space, we apply an off-policy program search process [Sutton *et al.*, 2000] to facilitate the learning of the semantic parser. Denote $\mathbb{P}(s)$ as the set of all valid programs in the CLEVR DSL for the input question s. We want to compute the gradient w.r.t. Θ_s, the parameters of the semantic parser:

$$\nabla_{\Theta_s} = \nabla_{\Theta_s} \mathbb{E}_P[r \cdot \log \Pr[P]],$$

where $P \sim \text{SemanticParse}(s; \Theta_s)$. In REINFORCE, we approximate this gradient via Monte Carlo sampling.

An alternative solution is to exactly compute the gradient. Note that in the definition of the reward r, only the set of programs $\mathbb{Q}(s)$ leading to the correct answer will contribute to the gradient term. With the perception module fixed, the set $\mathbb{Q}$ can be efficiently determined by an off-policy exhaustive search of all possible programs $\mathbb{P}(s)$. In the third stage of the curriculum learning, we search for the set $\mathbb{Q}$ offline based on the quantified results of concept classification and compute the exact gradient $\nabla\Theta_s$. An intuitive explanation of the off-policy search is that, we enumerate all possible programs, execute them on the visual representation, and find the ones leading to the correct answer. We use $\mathbb{Q}(s)$ as the "groundtruth" program annotation for the question, to supervise the learning, instead of running the Monte Carlo sampling-based REINFORCE.

Spurious program suppression. However, directly using $\mathbb{Q}(s)$ as the supervision by computing $\ell = \sum_{p \in \mathbb{Q}(S)} - \log \Pr(p)$ can be problematic, due to the spuriousness or the ambiguity of the programs. This comes from two aspects:
1) intrinsic ambiguity: two programs are different but equivalent. For example

P1: `AEQuery(Color, Filter(Cube), Filter(Sphere))` and
P2: `Exist(Filter(Sphere, AERelate(Color, Filter(Cube))))`

are equivalent.
2) extrinsic spuriousness: one of the program is incorrect, but also leads to the correct

answer in a specific scene. For example,

P1: `Filter(Red, Relate(Left, Filter(Sphere)))` and
P2: `Filter(Red, Relate(Left, Filter(Cube)))`

may refer to the same red object in a specific scene. Motivated by the REINFORCE process, to suppress such spurious programs, we use the loss function:

$$\ell = \sum_{p \in \mathbb{Q}} \text{stop_gradient}(\Pr[p]) \cdot (-\log \Pr[p]).$$

The corresponding gradient ∇_{Θ_s} is,

$$\nabla_{\Theta_s} = \sum_{p \in \mathbb{Q}} \Pr[p] \cdot \nabla_{\Theta_s} (r \cdot \log \Pr[P]) = \nabla_{\Theta_s} \left(\sum_{p \in \mathbb{Q}} r \cdot \Pr[p] \right).$$

The key observation is that, given a sufficiently large set of scenes, a program can be identified as spurious if there exists at least one scene where the program leads to a wrong answer. As the training goes, spurious programs will get less update due to the sampling importance term $\Pr[p]$ which weights the likelihood maximization term.

F.5 Curriculum Learning Setup

During the whole training process, we gradually add more visual concepts and more complex question examples into the model. Summarized in Fig. 6-4(A), in general, the whole training process is split into 3 stages. First, we only use questions from lesson 1 to let the model learn object-level visual concepts. Second, we train the model to parse simple questions and to learn relational concepts. In this step, we freeze the neural operators and concept embeddings of object-level concepts. Third, the model gets trained on the full question set (lesson 3), learning to understand questions of different complexities and various format. For the first several iterations in this step, we freeze the parameters in the perception modules. In addition, during the training of all stages, we gradually increase the number of objects in the scene: from 3 to 10.

We select questions for each lesson in the curriculum learning by their depth of the latent program layout. For eaxmple, the program "`Query(Shape, Filter(Red, Scene))`" has the depth of 3, while the program "`Query(Shape, Filter(Cube, Relate(Left, Filter(Red, Scene))))`" has the depth of 5. Since we have fused consecutive `Filter` operations into a single one, the maximum depth of all programs is 9 on the CLEVR dataset. We now present

the detailed split of our curriculum learning lessons:

For lesson 1, we use only programs of depth 3. It contains three types of questions: querying an attribute of the object, querying the existence of a certain type of objects, count a certain type of objects, and querying if two objects have the same attribute (e.g., of the same color). These questions are almost about fundamental object-based visual concepts. For each image, we generate 5 questions of lesson 1.

For lesson 2, we use programs of depth less than 5, containing a number of questions regarding relations, such as querying the attribute of an object that is left of another object. We found that in the original CLEVR dataset, all `Relate` operations are followed by a `Filter` operation. This setup degenerates the performance of the learning of relational concepts such as `Left`. Thus, we add a new question template into the original template set: `Count(Relate(` $\cdot$ `, Filter(` $\cdot$ `, Scene))`) (e.g., "What's the number of objects that are left of the cube?"). For each image, we generate 5 questions of lesson 2.

For lesson 3, we use the full CLEVR question set.

Curriculum learning is crucial for the learning of NS-CL. We found that by removing the curriculum setup w.r.t. the number of object in the scenes, the visual perception module will get stuck at an accuracy that is similar to a random-guess model, even if we only use stage-1 questions. If we remove the curriculum setup w.r.t. the complexity of the programs, the joint training of the visual perception module and the semantic parser can not converge.

F.6 Ablation Study

We conduct ablation studies on the accuracy of semantic parsing, the impacts of the ImageNet pre-training of visual perception modules, the data efficiency of our model, and the usage of object-based representations.

F.6.1 Semantic parsing accuracy.

We evaluate how well our model recovers the underlying programs of questions. Due to the intrinsic equivalence of different programs, we evaluate the accuracy of programs by executing them on the ground-truth annotations of objects. Invalid or ambiguous programs are also considered as incorrect. Our semantic parser archives $>$ 99.9% QA accuracy on the validation split.

F.6.2 Impacts of the ImageNet pre-training.

The only extra supervision of the visual perception module comes from the pre-training of the perception modules on ImageNet [Deng *et al.*, 2009]. To quantify the influence of this

pre-training, we conduct ablation experiments where we randomly initialize the perception module following He *et al.* [2015]. The classification accuracies of the learned concepts almost remain the same except for `Shape`. The classification accuracy of `Shape` drops from 98.7 to 97.5 on the validation set while the overall QA accuracy on the CLEVR dataset drops to 98.2 from 98.9. We speculate that large-scale image recognition dataset can provide prior knowledge of shape.

F.6.3 Data Efficiency and Object-based Representations

In this section, we study whether and how the number of training samples and feature representations affect the overall performance of various models on the CLEVR dataset. Specifically, we compare the proposed NS-CL against two strong baselines: TbD [Mascharka *et al.*, 2018] and MAC [Hudson and Manning, 2018].

Baselines. For comparison, we implement two variants of the baseline models: TbD-Object and MAC-Object. Inspired by Anderson *et al.* [2018], instead of using a 2D convolutional feature map, TbD-Object and MAC-Object take a stack of object features as inputs, whose shape is $k \times d_{obj}$. k is the number of objects in the scene, and d_{obj} is the feature dimension for a single object. In our experiments, we fix $k = 12$ as a constant value. If there are fewer than 12 objects in the scene, we add "null" objects whose features are all-zero vectors.

We extract object features in the same way as NS-CL. Features are extracted from a pre-trained ResNet-34 network before the last residual block for a feature map with high resolution. For each object, its feature is composed of two parts: region-based (by RoI Align) and image-based features. We concatenate them to represent each object. As discussed, the inclusion of the representation of the full scene is essential for the inference of relative attributes such as size or spatial position on the CLEVR domain.

TbD and MAC networks are originally designed to use image-level attention for reasoning. Thus, we implement two more baselines: TbD-Mask and MAC-Mask. Specifically, we replace the original attention module on images with a mask-guided attention. Denotes the union of all object masks as M. Before the model applies the attention on the input image, we multiply the original attention map computed by the model with this mask M. The multiplication silences the attention on pixels that are not part of any objects.

Results. Table 6.3 summarizes the results. We found that TbD-Object and MAC-Object approach show inferior results compared with the original model. We attribute this to the design of the network architectures. Take the `Relate` operation (e.g., finds all objects left of a specific object x) as an example. TbD uses a stack of dilated convolutional layers to

propagate the attention from object x to others. In TbD-Object, we replace the stack of 2D convolutions by several 1D convolution layers, operating over the $k \times d_{obj}$ object features. This ignores the equivalence of objects (the order of objects should not affect the results). In contrast, MAC networks always use the attention mechanism to extract information from the image representation. This operation is invariant to the order of objects, but is not suitable for handling quantities (e.g., counting objects).

As for TbD-Mask and MAC-Mask, although the mask-guided attention does not improve the overall performance, we have observed noticeably faster convergence during model training. TbD-Mask and MAC-Mask leverage the prior knowledge of object masks to facilitate the attention. Such prior has also been verified to be effective in the original TbD model: TbD employs an attention regularization during training, which encourages the model to attend to smaller regions.

In general, NS-CL is more data-efficient than MAC networks and TbD. Recall that NS-CL answers questions by executing symbolic programs on the learned visual concepts. Only visual concepts (such as `Red` and `Left`) and the interpretation of questions (how to translate questions into executable programs) need to be learned from data. In contrast, both TbD and MAC networks need to additionally learn to execute (implicit or explicit) programs such as counting.

For the experiments on the full CLEVR training set, we split 3,500 images (5% of the training data) as the hold-out validation set to tune the hyperparameters and select the best model. We then apply this model to the CLEVR validation split and report the testing performance. Our model reaches an accuracy of 99.2% using the CLEVR training set.

F.7 Extending to Other Scene and Language Domains

F.7.1 Minecraft Dataset

We also extend the experiments to a new reasoning testbed: Minecraft worlds [Yi *et al.*, 2018]. The Minecraft reasoning dataset differs from CLEVR in both visual appearance and question types. Figure F-1 gives an example instance from the dataset.



Q: What direction is the closest creature facing?
A: Left.
P: Query(Direction, FilterMost(Closest,
Filter(Creature)
))

Figure F-1: **Examples from the Minecraft dataset.**

Setup. Following Yi *et al.* [2018], we generate 10,000 Minecraft scenes using the officially open-sourced tools by Wu *et al.* [2017]. Each image contains 3 to 6 objects. The objects are chosen from 12 categories, with 4 different facing directions (front, back, left and right). They stand on a 2D plane.

Besides different 3D visual appearance and image contexts, the Minecraft reasoning dataset introduces two new types of reasoning operations. We add them to our domain-specific language:

1. `FilterMost`(ObjectSet, Concept) → ObjectSet: Given a set of objects, finds the "most" one. For example, `FilterMost`(`Closest`, set) locates the object in the input set that is cloest to the camera (e.g., what is the direction of the closest animal?)

2. `BelongTo`(Object, ObjectSet) → Bool: Query if the input object belongs to a set.

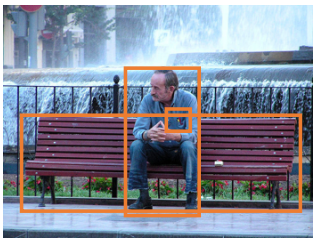
Results. Table F.5 summarizes the results and Figure F-4 shows sample execution traces. We compare our method against the NS-VQA baseline [Yi *et al.*, 2018], which uses strong supervision for both scene representation (e.g., object categories and positions) and program traces. In contrast, our method learns both by looking at images and reading question-answering pairs. NS-CL outperforms NS-VQA by 5% in overall accuracy. We attribute the inferior results of NS-VQA to its derendering module. Because objects in the Minecraft world usually occlude with each other, the detected object bounding boxes are inevitably noisy. During the training of the derendering module, each detected bounding box is matched with one of the ground-truth bounding boxes and uses its class and pose as supervision. Poorly localized bounding boxes lead to noisy labels and hurt the accuracy of the derendering module. This further influences the overall performance of NS-VQA.

F.7.2 VQS Dataset

We conduct experiments on the VQS dataset [Gan *et al.*, 2017]. VQS is a subset of the VQA 1.0 dataset [Antol *et al.*, 2015]. It contains questions that can be visually grounded: each question is associated with multiple image regions, annotated by humans as necessary for answering the question.

Setup. All models are trained on the first 63,509 images of the training set, and tested on the test split. For hyper-parameter tuning and model selection, the rest 5,000 images from the training set are used for validation. We use the multiple-choice setup for VQA: the models choose their most confident answer from 18 candidate answers for each question.

To obtain the latent programs from natural languages, we use a pre-trained syntactic dependency parser [Andreas *et al.*, 2016a; Schuster *et al.*, 2015] for extracting programs



Q: Does this man have any pens on him?
A: Yes.
P: Exist(Filter(Man,
Relate(Have, Filter(Pen))
))

Figure F-2: **Examples from the VQS dataset.** The orange bounding boxes are object proposals. On the right, we show the original question and answer in natural language, as well as the latent program recovered by our parser. To answer this question, models are expected to attend to the man and his pen in the pocket.

and concepts that need to be learned. A sample question and the program obtained by our parser is shown in Figure F-2. The concept embeddings are initialized by the bag of words (BoW) over the GloVe word embeddings [Pennington *et al.*, 2014].

Baselines. We compare our model against two representative baselines: MLP [Jabri *et al.*, 2016] and MAC [Hudson and Manning, 2018].

MLP is a standard baseline for visual-question answering, which treats the multiple-choice task as a ranking problem. For a specific candidate answer, a multi-layer perceptron (MLP) model is used to encode a tuple of the image, the question, and the candidate answer. The MLP outputs a score for each tuple, and the answer to the question is the candidate with the highest score. We encode the image with a ResNet-34 pre-trained on ImageNet and use BoW over the GloVe word embeddings for the question and option encoding.

We slightly modify the MAC network for the VQS dataset. For each candidate answer, we concatenate the question and the answer as the input to the model. The MAC model outputs a score from 0 to 1 and the answer to the question is the candidate with the highest score. The image features are extracted from the same ResNet-34 model.

Results. Table 6-7 summarizes the results. NS-CL achieves comparable results with the MLP baseline and the MAC network designed for visual reasoning. Our model also brings transparent reasoning over natural images and language. Example execution traces generated by NS-CL are shown in Figure F-5. Besides, the symbolic reasoning process helps us to inspect the model and diagnose the error sources. See the caption for details.

F.8 Visualization of execution traces and visual concepts

Another appealing benefit is that our reasoning model enjoys full interpretability. Figure F-3, Figure F-4, and Figure F-5 show NS-CL's execution traces on CLEVR, Minecraft, and VQS,

respectively. As a side product, our system detects ambiguous and invalid programs and throws out exceptions. As an example (Figure F-3), the question "What's the color of the cylinder?" can be ambiguous if there are multiple cylinders or even invalid if there are no cylinders.

Figure F-6 and Figure F-7 include qualitative visualizations of the concepts learned from the CLEVR and Minecraft datasets, including object categories, attributes, and relations. We choose samples from the validation or test split of each dataset by generating queries of the corresponding concepts. We set a threshold to filter the returned images and objects. For quantitative evaluations of the learned concepts on the CLEVR dataset, please refer to Table 6.2 and Table 6.5.

Operation	Signature	Semantics
`Scene`	() ⟶ ObjectSet	Return all objects in the scene.
`Filter`	(ObjectSet, ObjConcept) ⟶ ObjectSet	Filter out a set of objects having the object-level concept (e.g., red) from the input object set.
`Relate`	(Object, RelConcept) ⟶ ObjectSet	Filter out a set of objects that have the relational concept (e.g., left) with the input object.
`AERelate`	(Object, Attribute) ⟶ ObjectSet	(Attribute-Equality Relate) Filter out a set of objects that have the same attribute value (e.g., same color) as the input object.
`Intersection`	(ObjectSet, ObjectSet) ⟶ ObjectSet	Return the intersection of two object sets.
`Union`	(ObjectSet, ObjectSet) ⟶ ObjectSet	Return the union of two object sets.
`Query`	(Object, Attribute) ⟶ ObjConcept	Query the attribute (e.g., color) of the input object.
`AEQuery`	(Object, Object, Attribute) ⟶ Bool	(Attribute-Equality Query) Query if two input objects have the same attribute value (e.g., same color).
`Exist`	(ObjectSet) ⟶ Bool	Query if the set is empty.
`Count`	(ObjectSet) ⟶ Integer	Query the number of objects in the input set.
`CLessThan`	(ObjectSet, ObjectSet) ⟶ Bool	(Counting LessThan) Query if the number of objects in the first input set is less than the one of the second set.
`CGreaterThan`	(ObjectSet, ObjectSet) ⟶ Bool	(Counting GreaterThan) Query if the number of objects in the first input set is greater than the one of the second set.
`CEqual`	(ObjectSet, ObjectSet) ⟶ Bool	(Counting Equal) Query if the number of objects in the first input set is the same as the one of the second set.

Table F.1: **All operations in the domain-specific language for CLEVR VQA.**

Type	Example	Semantics
ObjConcept	`Red`, `Cube`, etc.	Object-level concepts.
Attribute	`Color`, `Shape`, etc.	Object-level attributes.
RelConcept	`Left`, `Front`, etc.	Relational concepts.
Object	●	A single object in the scene.
ObjectSet	{●, ■}	A set of objects in the scene.
Integer	$0, 1, 2, \cdots$	A single integer.
Bool	`True`, `False`	A single boolean value.

Table F.2: **The type system of the domain-specific language for CLEVR VQA.**

Step	Inputs	Outputs	Recursive Invocation
1	f_0	$\texttt{OpDecoder}(f_0) \rightarrow \texttt{Query}$; $\texttt{ConceptDecoder}(f_0) \rightarrow <$ Attribute 1 $>$; $\texttt{OEncoder}_0(f_0, \texttt{Query}) \rightarrow f_1$	$parse(f_1)$
2	f_1	$\texttt{OpDecoder}(f_1) \rightarrow \texttt{Filter}$; $\texttt{ConceptDecoder}(f_1) \rightarrow <$ ObjConcept 1 $>$; $\texttt{OEncoder}_0(f_1, \texttt{Filter}) \rightarrow f_2$	$parse(f_2)$
3	f_2	$\texttt{OpDecoder}(f_2) \rightarrow \texttt{Relate}$; $\texttt{ConceptDecoder}(f_2) \rightarrow <$ RelConcept 1 $>$; $\texttt{OEncoder}_0(f_2, \texttt{Relate}) \rightarrow f_3$	$parse(f_3)$
4	f_3	$\texttt{OpDecoder}(f_3) \rightarrow \texttt{Filter}$; $\texttt{ConceptDecoder}(f_3) \rightarrow <$ ObjConcept 2 $>$; $\texttt{OEncoder}_0(f_3, \texttt{Filter}) \rightarrow f_4$	$parse(f_4)$
5	f_4	$\texttt{OpDecoder}(f_3) \rightarrow \texttt{Scene}$;	(End of branch.)

Table F.3: **A step-by-step running example of the recursive parsing procedure.** The parameter $\{c_i\}$ is omitted for better visualization.

Signature	Implementation
`Scene`() → *out*: ObjectSet	$out_i := 1$
`Filter`(*in*: ObjectSet, *oc*: ObjConcept) → *out*: ObjectSet	$out_i := \min(in_i, \text{ObjClassify}(oc)_i)$
`Relate`(*in*: Object, *rc*: RelConcept) → *out*: ObjectSet	$out_i := \sum_j (in_j \cdot \text{RelClassify}(rc)_{j,i}))$
`AERelate`(*in*: Object, *a*: Attribute) → *out*: ObjectSet	$out_i := \sum_j (in_j \cdot \text{AEClassify}(a)_{j,i}))$
`Intersection`($in^{(1)}$: ObjectSet, $in^{(2)}$: ObjectSet) → *out*: ObjectSet	$out_i := \min(in_i^{(1)}, in_i^{(2)})$
`Union`($in^{(1)}$: ObjectSet, $in^{(2)}$: ObjectSet) → *out*: ObjectSet	$out_i := \max(in_i^{(1)}, in_i^{(2)})$
`Query`(*in*: Object, *a*: Attribute) → *out*: ObjConcept	$\Pr[out = oc] := \sum_i in_i \cdot \frac{\text{ObjClassify}(oc)_i \cdot b_a^{oc}}{\sum_{oc'} \text{ObjClassify}(oc')_i \cdot b_a^{oc'}}$
`AEQuery`($in^{(1)}$: Object, $in^{(2)}$: Object, *a*: Attribute) → *b*: Bool	$b := \sum_i \sum_j (in_i^{(1)} \cdot in_j^{(2)} \cdot \text{AEClassify}(a)_{j,i}))$
`Exist`(*in*: ObjectSet) → *b*: Bool	$b := \max_i in_i$
`Count`(*in*: ObjectSet) → *i*: Integer	$i := \sum_i in_i$
`CLessThan`($in^{(1)}$: ObjectSet, $in^{(2)}$: ObjectSet) → *b*: Bool	$b := \sigma((\sum_i in_i^{(2)} - \sum_i in_i^{(1)} - 1 + \gamma_c)/\tau_c)$
`CGreaterThan`($in^{(1)}$: ObjectSet, $in^{(2)}$: ObjectSet) → *b*: Bool	$b := \sigma((\sum_i in_i^{(1)} - \sum_i in_i^{(2)} - 1 + \gamma_c)/\tau_c)$
`CEqual`($in^{(1)}$: ObjectSet, $in^{(2)}$: ObjectSet) → *b*: Bool	$b := \sigma((-\lvert \sum_i in_i^{(1)} - \sum_i in_i^{(2)} \rvert + \gamma_c)/(\gamma_c \cdot \tau_c))$

Table F.4: **All operations in the domain-specific language for CLEVR VQA.** $\gamma_c = 0.5$ and $\tau_c = 0.25$ are constants for scaling and shift the probability. During inference, one can quantify all operations as Yi *et al.* [2018].

Model	Overall	Count	Exist	Belong	Query
NS-VQA	87.7	83.3	91.5	91.1	86.4
NS-CL	**93.3**	**91.3**	**95.6**	**93.9**	**94.3**

Table F.5: **Results on the Minecraft VQA dataset.** Our model achieves comparable results on the Minecraft dataset with baselines trained by full program annotations.

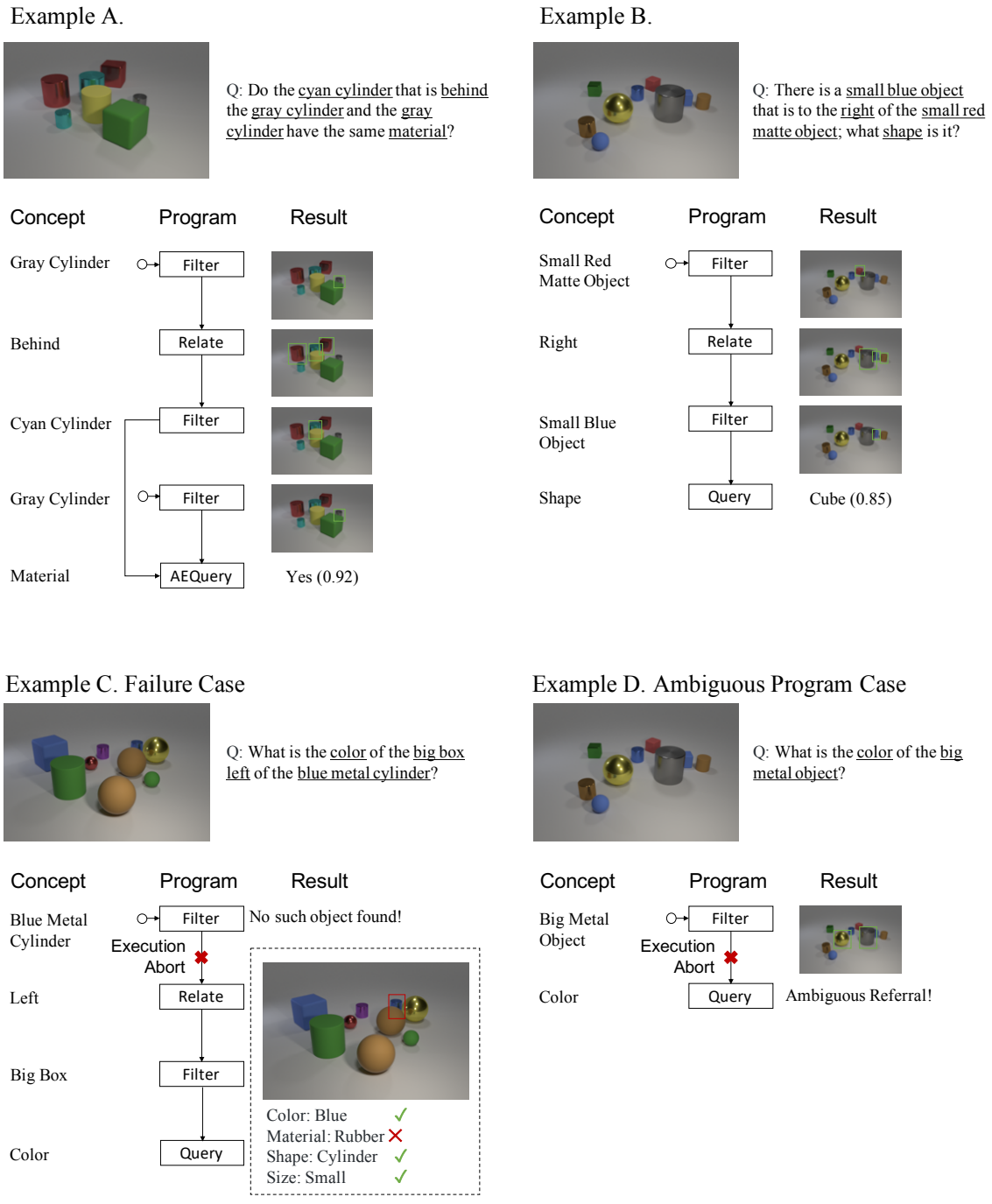


Figure F-3: **Visualization of the execution trace generated by NS-CL on the CLEVR dataset.** Example A and B are successful executions that generate correct answers. In example C, the execution aborts at the first operator. To inspect the reason why the execution engine fails to find the corresponding object, we can read out the visual representation of the object, and locate the error source as the misclassification of the object material. Example D shows how our symbolic execution engine can detect invalid or ambiguous programs during the execution by performing sanity checks.

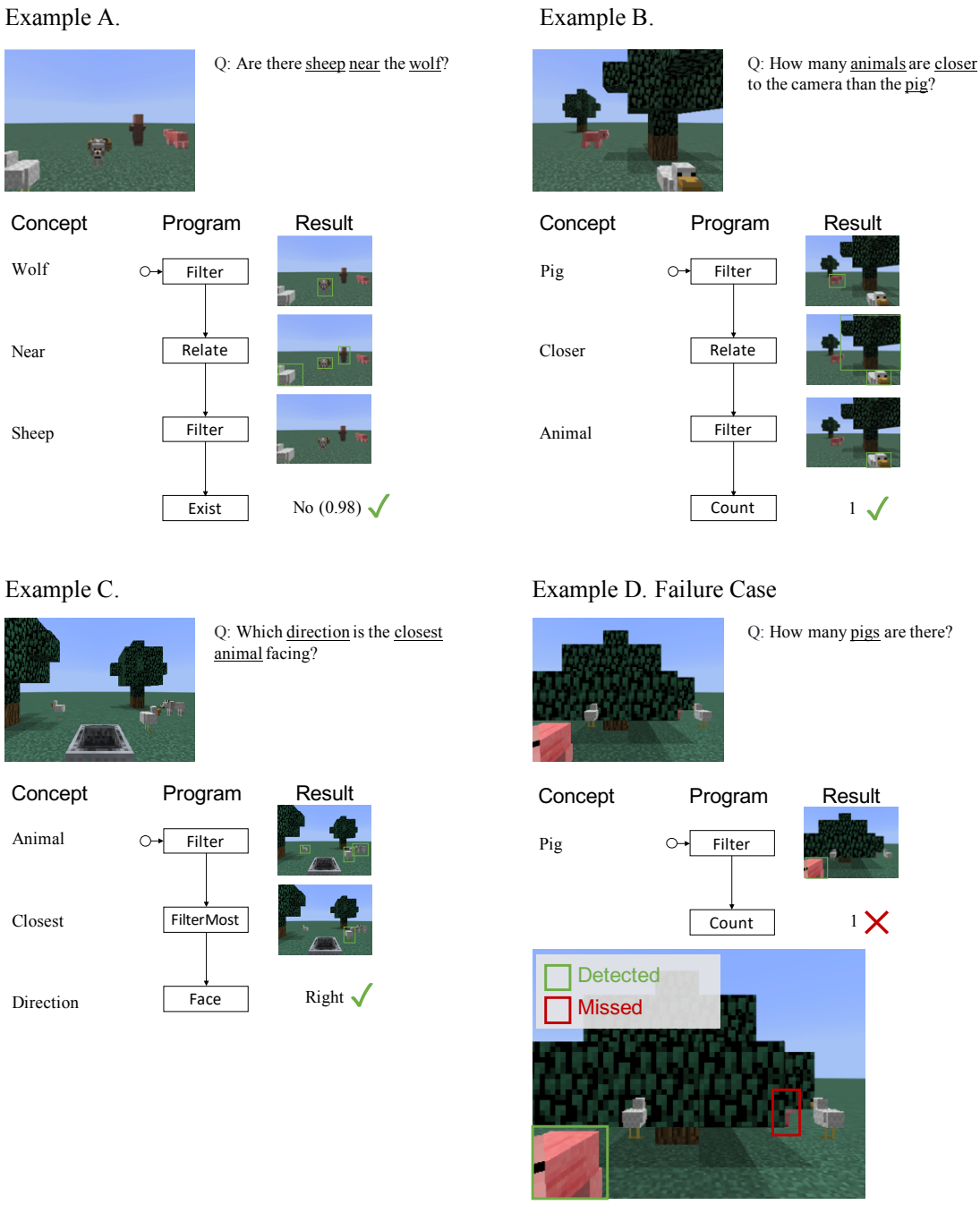


Figure F-4: **Visualization of the execution trace generated by NS-CL on the Minecraft VQA dataset.** Example A, B and C are successful execution. Example C demonstrates the semantics of the FilterMost operation. Example D shows a failure case: the detection model fails to detect a pig hiding behind the big tree.

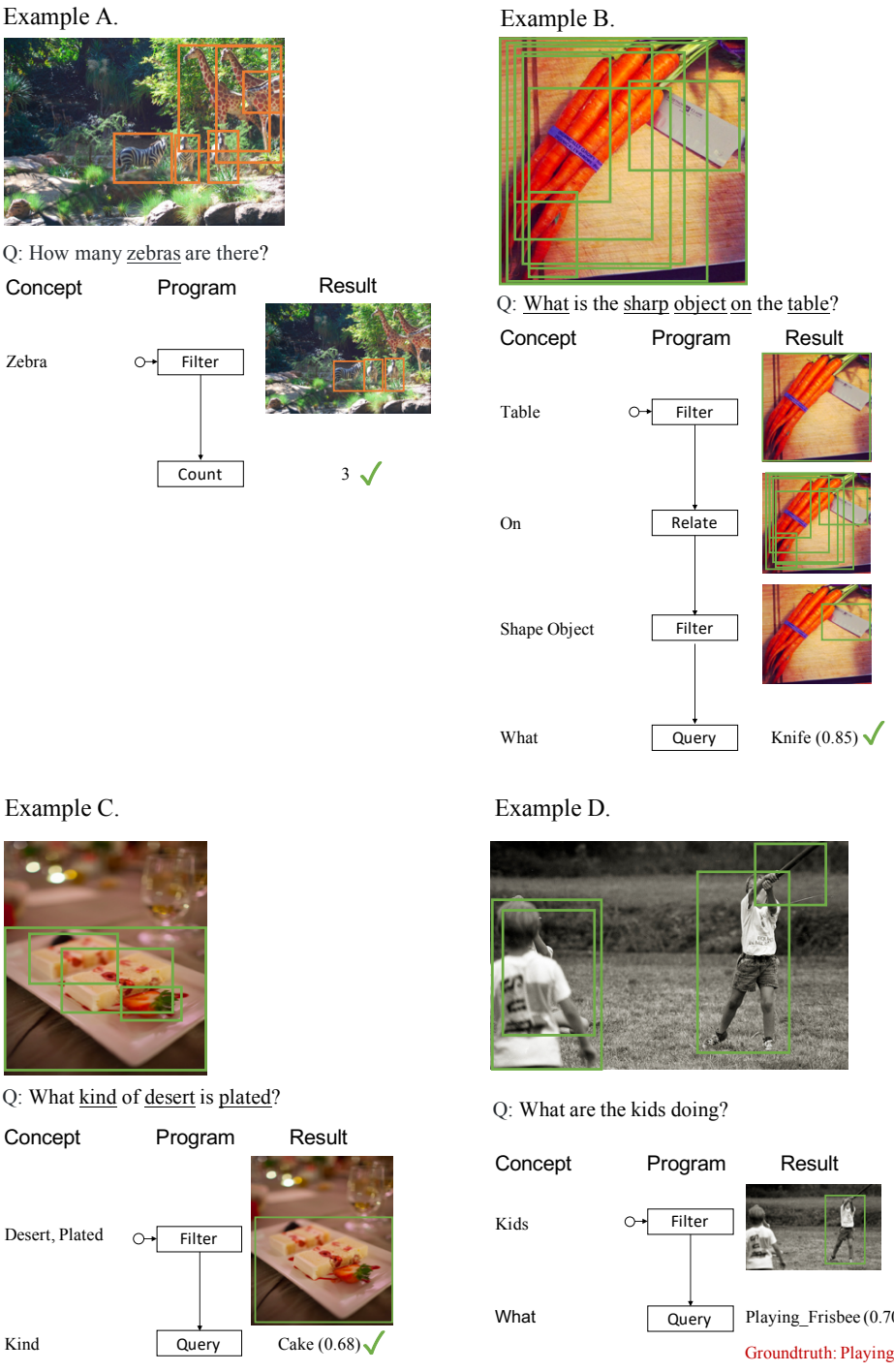
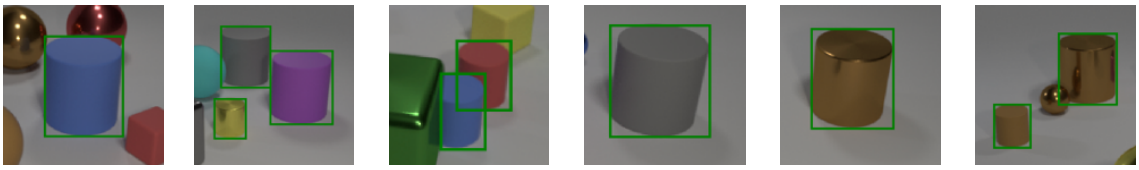
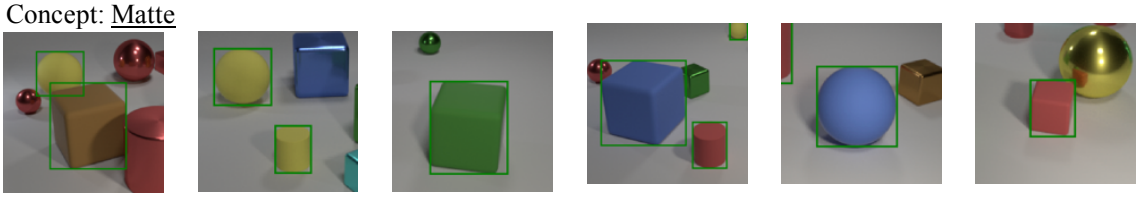


Figure F-5: **Visualization of the execution trace generated by NS-CL on the VQS dataset.** Execution traces A and B shown in the figure leads to the correct answer to the question. In Example C, although NS-CL answers the question correctly, it locates the wrong object during reasoning. In Example D, our model misclassifies the sport as frisbee.

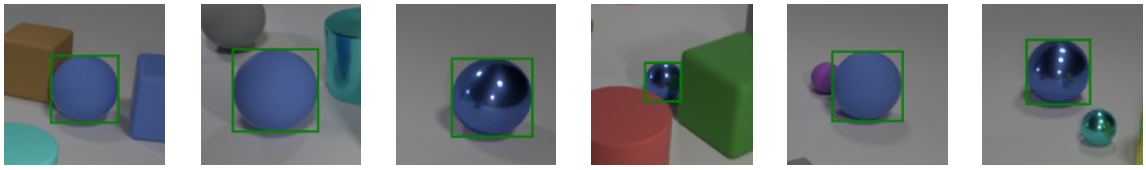
Concept: Cylinder



Concept: Matte



Concept: Blue Sphere



Concept: Yellow Object Left of Cylinder

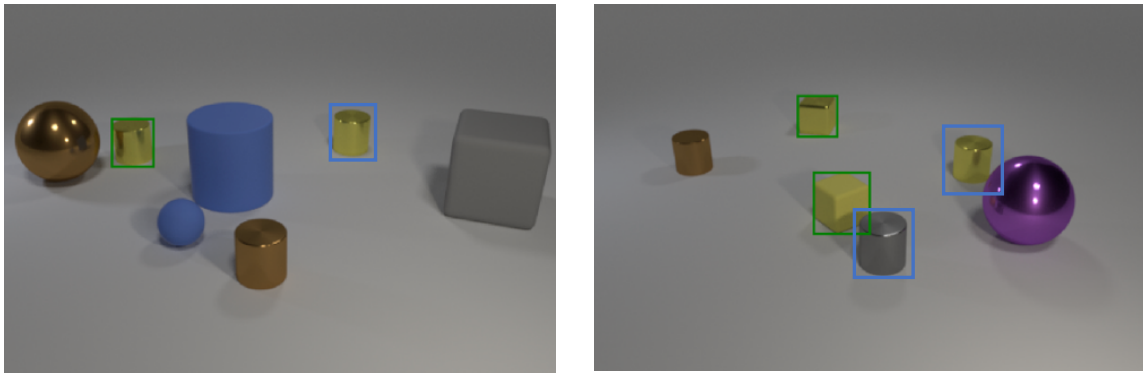
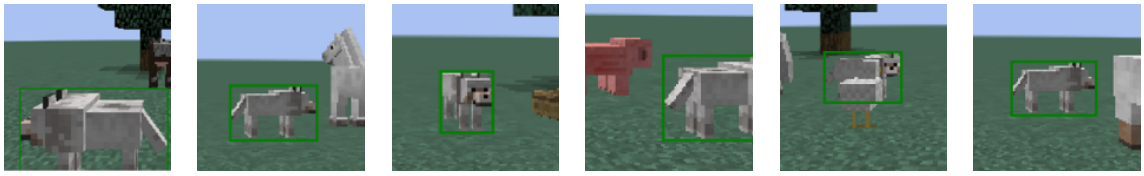
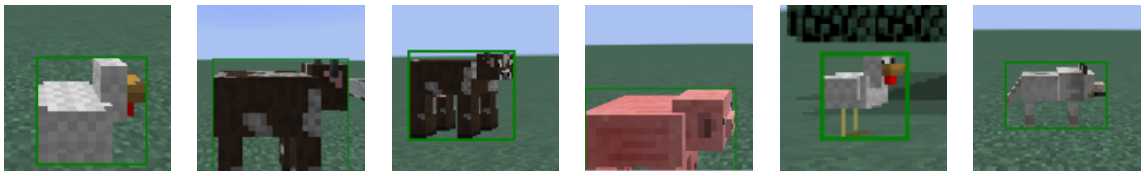


Figure F-6: **Concepts learned on the CLEVR dataset.**

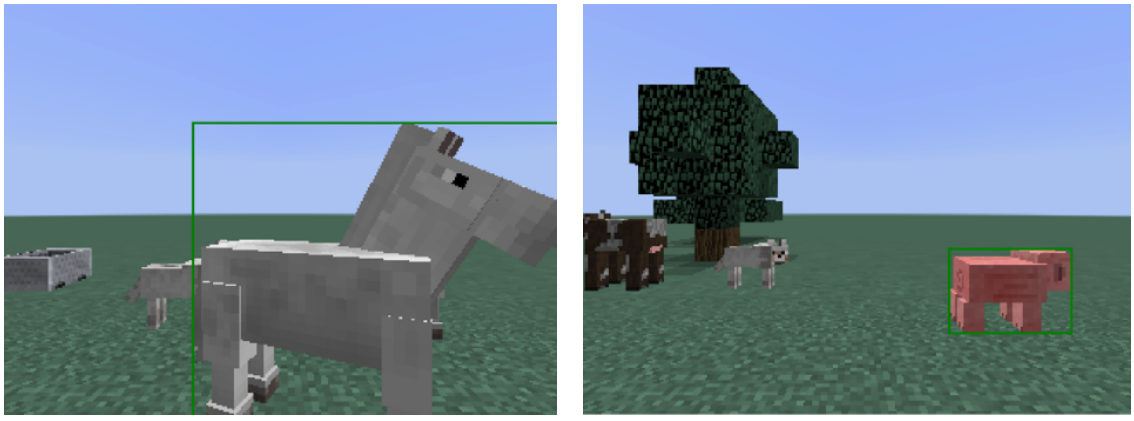
Concept: Wolf



Concept: Animal That Faces Right



Concept: Closest Living Thing



Concept: Villager Closer Than Pig

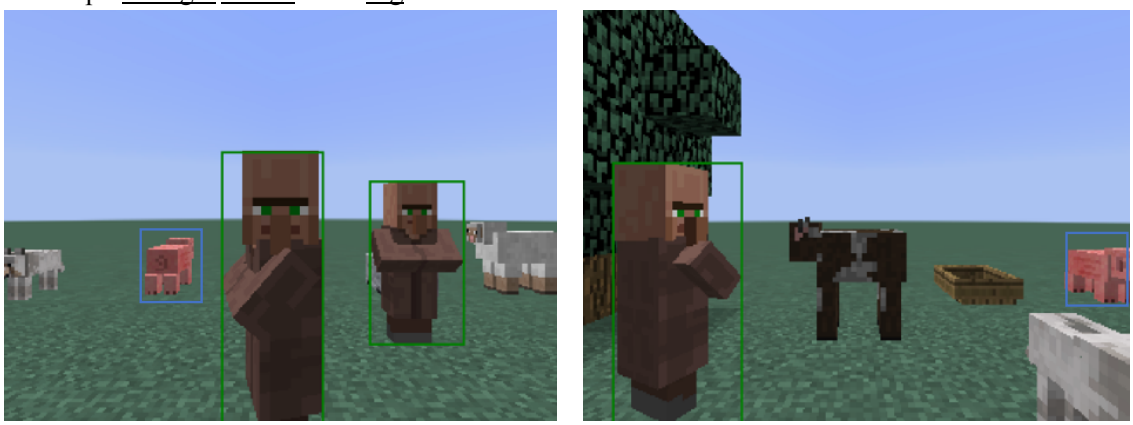


Figure F-7: **Concepts learned on the Minecraft dataset.**

Appendix G

Appendix for Fast Visual Concept Learning with Metaconcepts

This chapter is organized as the following. First, in Section G.1, we present our model details, especially the computation performed by our GNN- and RNN-based model for concept graphs and example graphs. Next, in Section G.2, we formally define the training objective for different stages in FALCON. Finally, in Section G.3 we supplement details and examples of our datasets and experimental setups.

G.1 Model Details

G.1.1 Remarks on the overall model structure

FALCON follows a modularized, concept-centric design pattern: the system maintains a set of concept embeddings, which naturally grows as it learns new concepts. We use separate modules for computing novel concept embeddings (Section 7.2.3) and processing queries (Section 7.2.2). These modules directly compute the novel concept embeddings, essentially classifiers on objects and their relationships, and use them to answer complex linguistic queries.

Here, all images and texts processed by the system will be "compressed" as these concept embeddings in a latent space. There are other possible designs, such as storing all training images and texts directly, and re-processing all training data for every single query in T_c. Despite being inefficient, as we will show in experiments, such design also shows inferior performance compared with FALCON.

G.1.2 Concept Embedding

Throughout the paper, We use a hidden dimension of $d = 100$ for the box embedding space (note that the vector representation for this box embedding space is 200-d since we have both centers and offsets), and $d = 512$ for the hyperplane and the hypercone embedding space. For the box embedding space, we set the smoothing hyperparameter to $\tau = 0.2$, following Li, Xiang and Vilnis, Luke and Zhang, Dongxu and Boratko, Michael and McCallum, Andrew [2019].

G.1.3 Prior Distribution

In both FALCON-G and FALCON-R, we assume a prior p_θ on concept embeddings, from which we sample the initial guess $e_c^{i,0}$. For box embedding space, we model the prior distribution of e_c so that its center and offset along all d dimensions can be generated by the same Dirichlet distribution parameterized by θ. In particular, we sample d independent samples $(x_{0p}, x_{1p}, x_{2p})_{p\in\{0,1,\cdots d-1\}}$ from a Dirichlet distribution parameterized by $\theta = (\theta_0, \theta_1, \theta_2)$. Then an initial guess for concept embedding of c is given as $\text{Cen}(c)_p = \frac{1}{2}(x_{1p} - x_{0p})$, and $\text{Off}(c)_p = \frac{1}{2}x_{2p}$. For hyperplane and hypercone embedding space, we model the prior as the same Gaussian distribution on all d dimensions, namely $e_c \sim \mathcal{N}(\mathbf{0}, \theta_0\mathbf{I}_d)$, where $\mathbf{I}$ is the identity matrix and θ_0 a scalar parameter. In both embedding space choices, θ can be learnt from back-propagation from $\mathcal{L}_{\text{meta}} = \mathcal{L}_{QA} - \lambda_{\text{prior}} \log p_\theta(e_c)$, and we set the hyperparameter $\lambda_{\text{prior}} = 1$.

G.1.4 FALCON-G

In FALCON-G, there two graphical neural networks, GNN_1 and GNN_2, which operates on the concept graph $G_{concept}$ and example graph $G_{example}$, respectively. Both of them are composed by stacking two graphical neural network (GNN) layers.

For convenience, we denote the input embedding of node i at layer $\ell \in \{0, 1\}$ as $h_i^{(\ell)}$. At layer ℓ, we first generate the message from node j to its neighbour i by:

$$m_{ji} = \text{MP}_k(h_j^{(\ell)}, h_i^{(\ell)}),$$

where k denote the relation between j and i, where MP is a message passing function, which we will formally define later.

Next, for each node i, we aggregate the message passed to i by computing an element-wise maximum (i.e., max pooling) of all neighbouring j's as $h_{\mathcal{N}_G(i)}^{(\ell)}$, where $\mathcal{N}_G(i)$ is the set of all neighboring nodes.

Finally, we update node embeddings for i based on its original embedding $h_i^{(\ell)}$ and aggregated messages $h_{\mathcal{N}_G(i)}^{(\ell)}$ for node i. Mathematically,

$$h_i^{(\ell+1)} = \text{UPDATE}(h_i^{(\ell)}, h_{\mathcal{N}_G(i)}^{(\ell)}).$$

Concretely, we implement the message passing model MP_k based on specific concept embedding spaces. For box embedding spaces, we first split $h_j^{(\ell)}$ as two vectors, denoted as $\text{Cen}(h_j^{(\ell)})$ and $\text{Off}(h_j^{(\ell)})$, respectively. Similarly, we split $h_i^{(\ell)}$ into $\text{Cen}(h_i^{(\ell)})$ and $\text{Off}(h_i^{(\ell)})$. For each hidden dimension $p \in \{0, 1, 2, \cdots, d-1\}$, we concatenate $\text{Cen}_p(h_j^{(\ell)}), \text{Off}_p(h_j^{(\ell)})$, $\text{Cen}_p(h_i^{(\ell)})$, and $\text{Off}_p(h_i^{(\ell)})$ as a 4-d vector, and applies a two-layer perceptron (MLP) on it. This MLP is shared across all hidden dimensions p. For different binary relations k, we use different MLPs. In our experiment, the aforementioned MLP has an output dimension of 10 and a hidden dimension of 20.

For the hyperplane and the hypercone embeddings, we simply concatenate $h_j^{(\ell)}$ and $h_i^{(\ell)}$, and send it into a MLP associated with relation k. The MLP has 1024 hidden units and 512 output units.

The update model UPDATE is implemented in a similar manner. For the box embedding space, we concatenate the hidden vector associated with each hidden dimension p of $h_i^{(\ell)}$ and $h_{\mathcal{N}_G(i)}^{(\ell)}$ as a 12-d vector. We send it to a feed-forward neural network with 20 hidden units and 4 output units, which is shared across d dimensions. If we denote the four output units as $o_{ip}^{(\ell)}, \delta_{ip}^{(\ell)}, o_{ip}^{\prime(\ell)}, \delta_{ip}^{\prime(\ell)}$, we use them to calculate the new box embedding $h_i^{(l+1)}$ as

$$\begin{aligned} \text{Cen}(h_{ip}^{(\ell+1)}) &= \text{Cen}(h_{ip}^{(\ell)}) + \sigma(o_{ip}^{(\ell)})\delta_{ip}^{(\ell)}; \\ \text{Off}(h_{ip}^{(\ell+1)}) &= \text{Off}(h_{ip}^{(\ell)}) + \sigma(o_{ip}^{\prime(\ell)})\delta_{ip}^{\prime(\ell)}, \end{aligned}$$

where σ is the sigmoid function. For hyperplane and hypercone embeddings spaces, we implement UPDATE by concatenate $h_i^{(\ell)}$ and $h_{\mathcal{N}_G(i)}^{(\ell)}$ and send it to a feed-forward neural network with 1024 hidden units and 1024 output units. If we denote its output as $o_{ip}^{(\ell)}, \delta_{ip}^{(\ell)}$, each having a length of 512, then the update model can be written as

$$h_i^{(\ell+1)} = h_i^{(\ell)} + \sigma(o_i) \odot \delta_i,$$

where $\odot$ denotes element-wise multiplication.

G.1.5 FALCON-R

In FALCON-R, concepts graph and example graphs are handled jointly in a unified framework. For each each object example $o_i^{(c)}$ associated with c, we treat their relationship as a "concept-example" relation. Thus, we can combine two graphs into a single graph. We denote the resulting graph as G^*.

We flatten the graph G^* by randomly order all relational tuples (c, c', rel) in G^*. We then feed them sequentially into an RNN cell. We implement the RNN cells RNN_1 and RNN_2 as a GNN used in FALCON-G, but is applied to a 2-node graph.

```
for j, i, k in G.edges:
    split = torch.stack([
        *h[j].chunk(2, -1),
        *h[i].chunk(2, -1)
    ], dim=-1)
    m[(j, i)] = mlp_1[k](split)
for i in G.nodes:
    h_N[i] = torch.stack([
        m[(jj, ii)] for (jj, ii) in m if i == ii
    ]).max(dim=0).values
for i in G.nodes:
    o, delta, o_prime, delta_prime = mlp_2(torch.stack([
        *h[i].chunk(2, dim=-1),
        *h_N[i].unbind(dim=-1)
    ])).unbind(dim=-1)
    h[i] = torch.cat([
        center + torch.sigmoid(o) * delta,
        offset + torch.sigmoid(o_prime) * delta_prime
    ], dim=-1)
```

Listing G.1: A snippet of one-step concept graph propagation based on the box embedding space. The code is written in PyTorch [Paszke *et al.*, 2019]. The code snippet updates the node embedding `h[i]` by first computing the messages passing from node `j` to node `i` in the first for-loop, aggregating the messages to node `i` by taking the element-wise max, and finally updating the node embedding of `i`.

Operation	Semantics
`Scene`	Return all objects in the scene.
`Filter`	Filter out the set of objects from input objects that have the specific object-level concept.
`Relate`	Filter out the set of objects that have the specific relational concept with as input object.
`AERelate`	Filter out the set of objects that have the same attribute value as the input object.
`Intersection`	Return the intersection of two object sets.
`Union`	Return the union of two object sets.
`Query`	Query the attribute of the input object.
`AEQuery`	Query if two objects have the same attribute value (e.g., color).
`Exist`	Query if an object set is empty.
`Count`	Return the size of an object set.
`CountLessThan`	Query if the size of the first object set is smaller than the size of the second object set.
`CountGreaterThan`	Query if the size of the first object set is greater than the size of the second object set.
`CountEqual`	Query if the size of the first set is the same as the size of the second object set.

Table G.1: **All program operations supported in CLEVR in FALCON.**

G.1.6 Semantic parser

In semantic parsing module, the sentence is translated into programs of primitive operations that have hierarchical layouts. For CLEVR dataset, the complete list of operation we provided is given in Table G.1. For CUB dataset, since there is only one bird in the image, we only provide operations `Scene`, `Filter` and `Exist` listed in G.1. Examples of the output of semantic parsing module are provided in Table G.2.

Our semantic parser follows the general framework of Seq2seq [Sutskever *et al.*, 2014] and our implementation follows NS-VQA [Yi *et al.*, 2018] which has been developed for VQA tasks. We first tokenize the input sentence into a sequence of word ids. Then, we feed the sequence into a Seq2Seq model and generate a sequence of $(op_i, concept_i)$, where $i = 1, 2, 3, ..., L_{out}$ where L_{out} is the output sequence length. op_i's are operations defined in the DSL, and they form a "flattened" program based on which we will recover a hierarchical program. Some operations such as *filter* has a concept input, in which case we use the corresponding $concept_i$. The semantic parser is trained on 10% of all generated data.

Dataset	Input	Output
CLEVR (Description Sentences)	The red objects that are both in front of the cyan object and behind the cube is a sphere.	`Filter(Intersection( Relate(Filter(Scene(), cyan), front), Relate(Filter(Scene(), cube), behind) ), red)`
CLEVR (Questions)	There is a rubber object to the left of the metal object; What is its color?	`Query(Filter( Relate(Filter(Scene(), metal), left), rubber), color)`
CUB (Description Sentences)	There is a frigatebird.	`Scene()`
CUB (Questions)	Is there a frigatebird?	`Exist(Filter(Scene(), frigatebird))`
GQA (Description Sentences)	The clouds are a white object.	`Filter(Scene(),clouds)`
GQA (Questions)	Is the standing man a red object?	`Exist(Filter(Filter( Filter(Scene(), standing), man), red))`

Table G.2: **Example questions and the programs recovered by the semantic parser in FALCON.** For description sentences, the semantic parser parse the referential expression in the sentence into a program. On CUB, since there is only one object in the image, the operation `Scene()` returns the only object in the image.

G.1.7 Neuro-Symbolic Progrm Execution

We mostly follow the original design of NS-CL as in Chapter 6. The neuro-symbolic executor executes the program based on both the scene representation and the concept embeddings. Lying in the core of execution is the representation of object sets, which are immediate results during the execution. Each set of objects (such as the output of a `Filter` operation, is a vector of length N, where N is the number of objects in the scene. Each entry is a score ranging from 0 to 1. We use the same implementation as NS-CL, with two modifications:

First, the probability of classifying object o as concept c, $\Pr[e_o \mid e_c]$ (`ObjClassify` in the NS-CL), is implemented based on different embedding spaces: box, hypercone and hyperplane. The original NS-CL uses the hypercone representation.

Second, the probability that two objects having the same attribute (named `AEClassify` in the NS-CL paper) are treated as special kinds of relational concepts: `has_same_color`, `has_same_shape`, etc. Thus, the `AERelate` operation is implemented in the same way as `Relate`.

G.1.8 Baseline Implementation

CNN+LSTM In CNN+LSTM, we use a ResNet-34 network with a global average pooling layer to extract the image representation. Each image feature is a feature vector of dimension 512. We concatenate the description sentences, supplemental sentences (if any), and test questions with a separator to form the text input. We use a bi-directional LSTM to encode the text into a 512-dimensional vector (we use the output state of the last token). We use a two-layer LSTM, each with a hidden state of dimension 512. The image representation of both images (the example image in X_c, and the test image in T_c), together with the text encoding, are sent into a multi-layer perceptron (MLP) to predict the final answer. The MLP has two layers, with a hidden dimension of 512.

MAC The MAC network takes an image representation (encoded by CNNs) and a text representation (encoded by RNNs) as its input, and directly outputs a distribution over possible answers. Following the original paper Hudson and Manning [2018], we use the first four residual blocks of a ResNet-101 He *et al.* [2015] to extract image representations, which is a 2D feature map with hidden dimension 512. The image representations of the example image and the test image an stacked channel-wise into a 2D feature map of hidden dimension 1024. The Similar to CNN+LSTM, we concatenate the description sentences, supplemental sentences (if any), and test questions with a separator to form the text input. For all other parts, we use the same setting as the original paper Hudson and Manning

[2018], such as the hidden dimension for all units in MAC.

NSCL+LSTM In NSCL+LSTM, we use the same semantic parsing module and program execution module to derive the example object embeddings $o_i^{(c)}$. These object embeddings are encoded by a bi-directional LSTM into a 512-dimensional vector. We use a separate bi-directional LSTM to encode the $G_{concept}$ into a 512-dimensional vector. Embeddings of related concept c', concatenated with an embedding for the relation *rel*, are fed in to the LSTM one at a time. Two LSTM both use as output the output state associated with the last input, and have two layers and 512 hidden dimension. These output of two LSTMs are concatenated and sent to a linear transform to predict the concept embedding e_c. The output of the linear transform e_c will be used by the neuro-symbolic program execution module to predict the final answer to queries. NSCL+LSTM is an ablative model of FALCON-R. It keeps similar designs as FALCON-R except it places visual examples and linguistic guidance on different streams, whereas FALCON-R uses a single recurrent mechanism to capture both the visual and linguistic information.

NSCL+GNN In NSCL+GNN, the supplemental sentence D_c is first parsed into conceptual graph $G_{concept}$. All nodes corresponding to related concepts c' are initialized by the concept embeddings of c'. The node corresponding to concept c is initialized from the "average" of all visual examples. For both hyperplane and hypercone embedding spaces, the average is computed by taking the mean of (L2-normalized) example object embeddings. For box embedding spaces, it is computed as the specific boundary (i.e., the minimal bounding box that covers all examples) of visual example embeddings. For meta-learning setting without supplmentary sentences, we use the average box size of all concepts in the training set to initialize the box size for novel concepts, so the NSCL+GNN model would become a special example of the Prototypical Network. The GNN has two layers, and has the same design as our mode FALCON-G. NSCL+GNN is an ablative model of FALCON-G. It keeps all designs of FALCON-G except that it directly computes the "average" of visual examples. In contrast, in FALCON-G, the example object information are gathered with a example graph neural network encoder.

G.2 Training Details

Our training paradigm consists of three stages: pre-training, meta-training and meta-testing. Fig. G-1 gives an illustrative overview of these three stages. Recall that we have processed the dataset by splitting all concepts into three groups: C_{base}, C_{val}, and C_{test}, and each concept in the dataset is associated with a 4-tuple (c, X_c, D_c, T_c).

In pre-training stage, our model is trained on all concepts c in base concept set C_{base}. The concept embedding of c and the visual representations are trained using question-answer pairs from T_c. This step is the same as the concept learning stage of NS-CL, except that we are using a geometric embedding space. In particular, based on the classification confidence scores of object properties and their relations, the model outputs a distribution over possible answers for each recovered program. On the CLEVR dataset, we use the same curriculum learning setting as NS-CL, based on the number of objects in the image and the complexity of the questions. In the meta-training stage, we fix the concept embedding for all concepts $c \in C_{base}$, and train the GNN modules (in FALCON-G) or the RNN modules (in FALCON-R) with our meta-learning objective $\mathcal{L}_{meta}$. In this step, we randomly sample task tuples (c, X_c, D_c, T_c) of all concepts in C_{base}, and use concepts in C_{val} for validation and model selection. In the meta-testing state, we evaluate each model by its downstream task performance on concepts drawn from C_{test}.

One issue with end-to-end methods (CNN+LSTM and MAC) is that they assume that the concept vocabulary is known. Upon seeing a new concept, they do not have the mechanism for inducing a new word embedding for this concept c. To address this issue, in the meta-training and meta-testing stages, the appearance of the novel concept c in all texts will be replaced by a special token <UNK>.

Our model is trained with the following hyperparameters. During the pre-training stage, the model is trained for 50000 iterations with a batch size of 10, using an Adam optimizer Kingma and Ba [2015] with learning rate 0.0001. During the meta-training stage, the model is trained for 10000 iterations with a batch size of 1 (one novel concept per iteration). We train the model with the same optimizer setup (Adam with a learning rate of 0.001). In both stages, we evaluate the model on their corresponding validation set every 1000 iterations. We decrease the learning rate by a factor of 0.1 when the validation accuracy fails to improve for 5 consecutive validation evaluations, i.e. the iteration with the best validation accuracy is more than 5 validations ago.

G.3 Experiment Details

G.3.1 The CUB Dataset

In the CUB dataset, descriptive sentences are generated from the referential expressions produced by the templates in Table G.3a, containing only concepts from C_{base}. We provided an approximate of 1.8k descriptive sentences, 1055 of which for C_{base}, 415 for C_{val} and 360 for C_{test}. Supplemental sentences are generated based on the templates in Table G.3b. In supplemental sentences for concept $c \in C_{base}$, we randomly remove a related concept c'

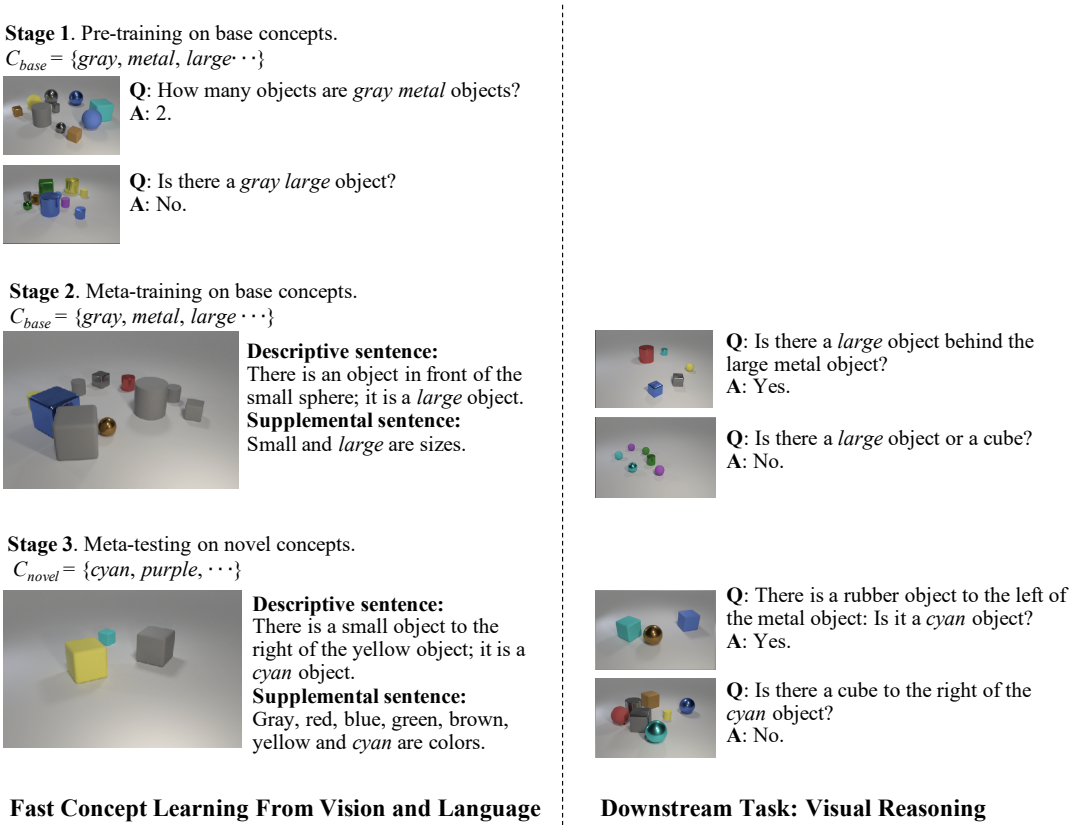


Figure G-1: **An overview of the overall training paradigm of FALCON.**

(a) The descriptive sentence template in CUB.	
Template	There is a/an <CONCEPT>.
Example	There is a white-eyed vireo.
Description	<CONCEPT> is the word representing concept c.
(b) The supplemental sentence template in CUB.	
Template	<CONCEPT-1>, <CONCEPT-2>$\cdots$ <CONCEPT-N> are a kind of <HYPERNYM>.
Example	Black-capped vireo, philadelphia vireo, warbling vireo, white-eyed vireo are a kind of vireos.
Description	<CONCEPT-i> are words representing c and c' that share a *cohypernym* relation with c. <HYPERNYM> denotes c's hypernym.
(c) The test question template in CUB.	
Template	Is there a/an <CONCEPT>.
Example	Is there a white-eyed vireo.
Description	<CONCEPT> is the word representing concept c.

Table G.3: **Templates for generating descriptive sentences, supplemental sentences and test questions in CUB.**

from the sentence with a probability of 0.2. We generate 54k testing questions based on the templates in Table G.3c, 31.5k of them paired with concepts in C_{train}, 12.5k of them paired with concepts in C_{val} and 10k of them paired with concepts in C_{test}. Descriptive sentences, supplemental sentences and testing questions are zipped together to form the input for fast concept learning task. The same template generates Pre-training questions as testing questions. We generated around 11.8k question-image pairs for pre-training. During training, we apply standard data augmentation techniques on the images, including horizontal flip, random resizing and cropping.

Extension: continual learning. In the continual learning setting, concepts that has been learned in the meta-testing instances can be used as supporting concepts for other concepts. We implement this idea as the following. Since CUB is constructed based on a bird taxonomy, we can define the distance between two concepts by the length of their shortest path on the bird taxonomy. We define the *hop* number of each concept as their distance with the closest concepts $c' \in C_{base}$. In the meta-testing stage, we test models with concepts of increasing *hop* numbers. For example, we first teach the model with 1-hop concepts, which directly relates to concepts they have seen in C_{base}. Next, we add 2-hop concepts. During the learning of 2-hop concepts, the concept embedding of those 1-hop concepts are fixed. The detailed result for concept-centric methods for continual concept learning can be found in Table G.4.

Model	Performance		
	Box	Hyperplane	Hypercone
NSCL+LSTM	75.21	67.78	71.06
NSCL+GNN	71.04	64.74	72.37
FALCON-G	**79.67**	73.14	64.37
FALCON-R	78.76	65.38	68.11

Table G.4: **The continual concept learning performance on the CUB dataset.**

Model	Performance		
	Box	Hyperplane	Hypercone
NSCL+LSTM	71.31	62.45	67.50
NSCL+GNN	72.53	61.56	71.25
FALCON-G	**75.28**	64.16	68.73
FALCON-R	75.01	62.78	67.96

Table G.5: **The continual concept learning performance with only visual examples on the CUB dataset.** This task is also referred as few-shot learning, where NSCL+GNN will fall back to a Prototypical Network [Snell *et al.*, 2017].

The accuracy is averaged over all 72 test concepts in C_{test}. For reference, we also show the results of applying these models to the fast concept learning tasks without supplemental sentences in Table G.5 (the few-shot learning setting). This results are computed over all 72 concepts in C_{test}, although we do not use any supplemental sentences. Our model makes the best use of supplemental sentences in inducing novel concept embeddings.

G.3.2 The CLEVR Dataset.

CLEVR dataset contains 15 individual concepts, 8 of them associates with colors, 3 associates with shapes, 2 associated with the materials of objects and 2 associated with the sizes of objects. Each split is generated by selecting 2 colors as validation concepts C_{val}, and another 2 colors and 1 shape as testing concepts C_{test}.

On CLEVR, descriptive sentences are generated based on the CLEVR-Ref+ [Liu *et al.*, 2019a] dataset. Shown in Table G.6a, we first generate a referential expression for the target object, and associate it with the novel concept c. The referential expression contains only concepts from C_{base}. In total we have generated 7.5k pairs of descriptive sentences and images, 5k of which are for C_{base}, 1k for C_{val}, and another 1.5k for C_{test}. supplemental sentences sentences are generated based on template in Table G.6b by combining the concept c and its related concepts c'. We generate 225k pairs of testing questions and images based

(a) The descriptive sentence template in CLEVR.	
Template	There is a <REFEXP>; it is <CONCEPT>.
Example	There is a small object to the right of the yellow object; it is a cyan object.
Description	<REFEXP> is a referential expression generated by CLEVR-Ref+ [Liu *et al.*, 2019a]. <CONCEPT> is the word for concept c.
(b) The supplemental sentence template in CLEVR.	
Template	<CONCEPT-1>,<CONCEPT-2>$\cdots$<CONCEPT-N> are <ATTRIBUTE>.
Example	Cube, sphere and cylinder are shapes.
Description	<CONCEPT-i> are words representing c and its related concepts c'. <ATTRIBUTE> is the word for an attribute of a object, one of *shape*, *color*, *material* and *size*.
(c) The test question generation template in CLEVR.	
Template	<QUEST>
Example	Does the small blue thing have the same material as the gray object?
Description	<QUEST> are question generated from the original CLEVR paper [Johnson *et al.*, 2017a].

Table G.6: **Templates for generating descriptive sentences, supplemental sentences and test questions in CLEVR.**

on the templates in Table G.6c. 150k of them are associated with concepts in C_{train}, 30k for C_{val}, and another 45k for in C_{test}. Testing question contains only one occurrence of concept c and possibly many occurrences of base concepts. An example of the generated texts can be found in Fig. G-1. We generated 402k questions for pre-training.

Learning from biased data. We supplement a concrete example to illustrate how our model can learn novel visual concepts from biased data. Shown in Fig. G-2, in the first task, the learner is trained on a novel concept *cylinder*. However, all training examples of *cylinders* are also *purple*. Thus, without any supplemental information about whether the novel concept is a color or a shape, the learning task itself is ambiguous. Our model is capable of resolve such kind of ambiguity by interpreting supplemental sentences. We have also included the full experimental result of this setup in Table G.7.

Handling non-prototypical concepts. On CLEVR, we have studied a specific type of concept-concept relationship: *hypernym*. It is worth noting that there is a difference between the *hypernym* relations in CUB dataset and *hypernym* relations in CLEVR dataset. Specifically, the hypernym of a bird category is also a bird category. However, the hypernym of a concept in CLEVR (e.g., color, shape, material, etc.) is no longer concepts that can be

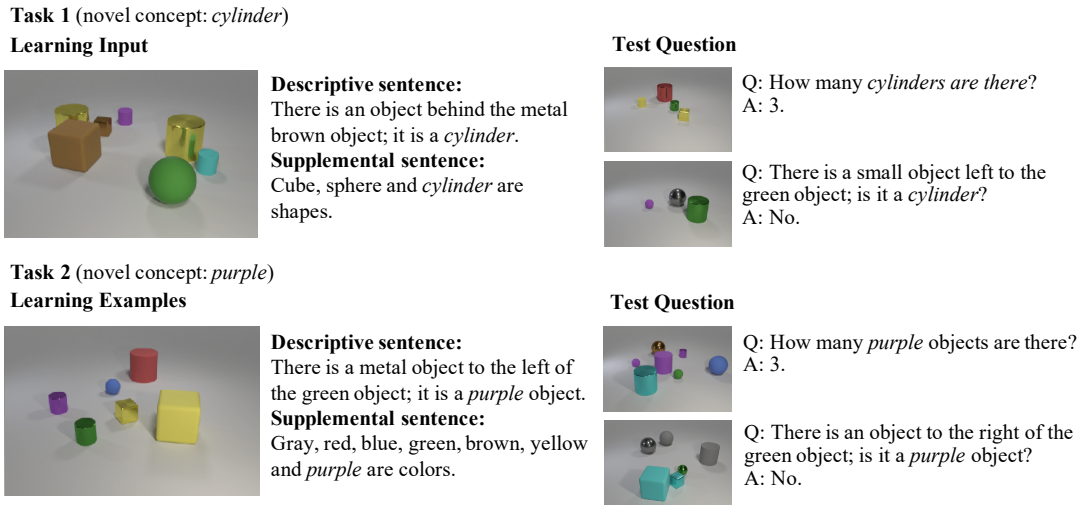


Figure G-2: **A visualization of fast concept learning from biased data with supplemental sentences on CLEVR.** Notice that in both tasks, the visual examples are purple cylinders. The association between the new concept and the visual appearance (shape or color) is ambiguous, without supplemental sentences.

Model	Example Only	Example+Supp.	Δ
CNN+LSTM	41.61	42.38	0.77
MAC	62.38	62.40	0.02
NSCL+LSTM	57.76	61.79	4.03
NSCL+GNN	64.75	84.83	20.08
FALCON-G	69.71	**87.29**	17.57
FALCON-R	69.70	86.21	16.50

Table G.7: **Ablated study of FALCON tasks w/. and w/o. supplemental sentences on the biased CLEVR dataset.** All concept-centric methods use the box embedding space.

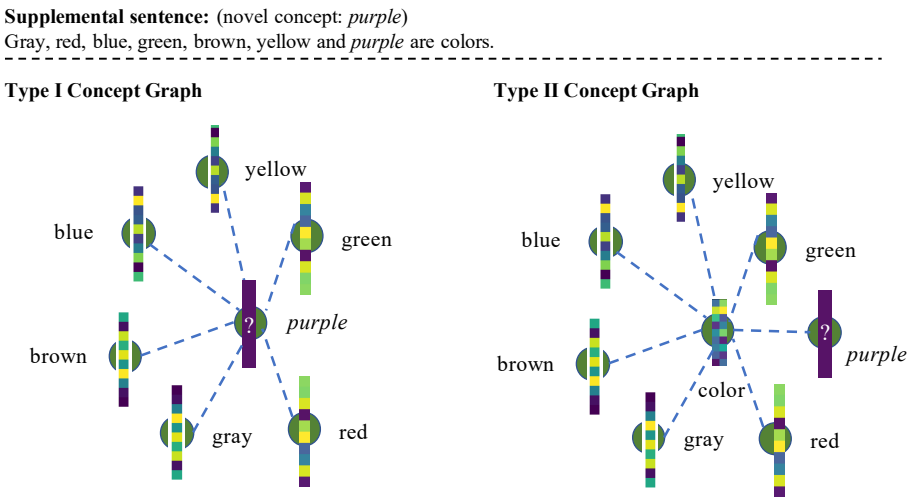


Figure G-3: **Two implementations of graph $G_{concept}$.**

visually grounded. They are non-prototypical concepts because they can not be associated with a prototypical visual instance.

We have studied two different ways to handle such kind of hypernym relation. They are different in terms of their way of constructing the concept graph $G_{concept}$, namely Type I and Type II. Illustrated in Fig. G-3, in Type I graph, the novel concept c = purple is connected with other concepts that share the same hypernym with c. In type II $G_{concept}$, the non-prototypical concept *color* is treated as a node in the concept graph. All concepts that are "colors", including c is connected to this new node *color*. Our experiment shows that, in the fast concept learning test with continual learning setting, our best model (FALCON-G + Box Embedding) achieves a test accuracy of 88.40% using the type I concept graph. With the type II concept graph, it can only reach the accuracy of 71.34%. Thus, throughout the paper, we will be using concept graphs of type I.

G.3.3 The GQA dataset

In the GQA dataset, descriptive sentences are generated from the referential expressions produced by the templates in Table G.8a, containing only concepts from C_{base}. We provided an approximate of 3.9k descriptive sentences, 2.3k of which for C_{base}, 600 for C_{val} and 1k for C_{test}. Supplemental sentences are generated based on the templates in Table G.3b. In supplemental sentences for concept $c \in C_{base}$, we randomly remove a related concept c' from the sentence with a probability of 0.2. We generate 117k testing questions based on the templates in Table G.3c, 13.8k of them paired with concepts in C_{train}, 18k of them paired with concepts in C_{val} and 30k of them paired with concepts in C_{test}. We generated around 13.4k question-image pairs for pre-training. We use the object-based features produced by a

(a) The descriptive sentence template in GQA.	
Template	The <CONCEPT-1> (<CONCEPT-2>) (object) is a/an <CONCEPT> (object).
Example	The small man is a sitting object.
Description	<CONCEPT> is the word for concept c. <CONCEPT-i> are other concepts.
(b) The supplemental sentence template in GQA.	
Template	<CONCEPT-1>,<CONCEPT-2>$\cdots$<CONCEPT-N> describes the same property of an object.
Example	Sitting, standing describes the same property of an object.
Description	<CONCEPT-i> are words representing c and its related concepts c'.
(c) The test question generation template in GQA.	
Template	Is the <CONCEPT-1> (<CONCEPT-1>) (object) a/an <CONCEPT> (object)?
Example	Is the white woman a sitting object?
Description	<CONCEPT> is the word for concept c. <CONCEPT-i> are other concepts.

Table G.8: **Templates for generating descriptive sentences, supplemental sentences and test questions in GQA.**

pretrained Faster-RCNN model [Ren *et al.*, 2015], which is provided in the dataset.

The main result is listed in Table G.9 and G.10. We also compare the model performance on the "few-shot" learning setting, where no supplemental sentences are provided. Out experiments shows that our model (FALCON-G or FALCON-R with box embedding spaces) outperforms other baselines in both settings.

G.3.4 The connections between the box embedding and our formulation

We justify the choice of the box embedding space from two aspects. First, intuitively some supplemental sentences in our setting can be modeled as concept entailment relationships, which can be well modeled by the box embedding space. Second, empirically, the box embedding space, among three embedding space choices studied in the paper, is the best choice for representing entailment.

Modeling concept entailments (intuition). In order to learn a new concept c, the model receives supplemental sentences relating c with known concepts c'. A specific type of relationship is the hypernym relation. The fact that c is a hypernym of c' implies that "all objects that belong to concept c' also belong to concept c". In the box embedding space, such relationship can be represented as a constraint that "the box of c is inside the box c'". Thus, the geometric structures of the box embedding space naturally fits the concept

Model	QA Accuracy		
CNN+LSTM	54.47		
MAC	54.99		
	Box	Hyperplane	Hypercone
NSCL+LSTM	55.41	54.19	54.81
NSCL+GNN	55.32	50.07	54.75
FALCON-G	**55.96**	54.22	54.56
FALCON-R	55.42	55.73	55.79

Table G.9: **Fast concept learning performance on the GQA dataset.** Our model FALCON-G and FALCON-R outperforms all baselines.

Model	QA Accuracy		
CNN+LSTM	48.73		
MAC	54.45		
	Box	Hyperplane	Hypercone
NSCL+LSTM	53.54	53.16	52.83
NSCL+GNN	50.15	50.74	52.03
FALCON-G	53.89	53.24	52.22
FALCON-R	**54.60**	52.86	52.37

Table G.10: **Fast concept learning with only visual examples, evaluated on the GQA dataset.** In this setting, NSCL+GNN will fallback to a Prototypical Network [Snell *et al.*, 2017].

entailment relations in our data.

There are two things to be noted. First, there are other non-entailment relations that can not be explicitly written down as a constraint. For example, the "same kind" relationship between concept `red` and concept `blue`: they are both colors. Second, instead of explicitly imposing these constraints while inferring the concept embedding for c, we use learned models to directly predict the embedding of c - but we hope neural models to leverage such geometric structures.

Comparison with other box embedding spaces (empirical support). We have already compared the "box embedding" space with other, commonly-used approaches, such as the "hyperplane space" (i.e., the score is computed by a dot-product), and the "hypercone space" (i.e., the score is computed by cosine-similarity) using downstream task performance measures. The box embedding space outperforms the other two (Table 1 and 2 of the main paper).

Other than downstream task performances, to further validate our intuition, we conduct further analysis on whether the predicted box embeddings of concepts actually satisfy the entailment relations. Specifically, using the CUB dataset, we use trained FALCON-G models based on different embedding spaces to predict the concept embeddings in the validation and the test concept set. For concepts in the training set, we use the learned concept embeddings in the pre-training stage. Next, we use all concept embeddings and the underlying conditional probability measure for each space to compute $\Pr[c'|c]$ for all pairs of concepts (c, c').

We denote $\Pr[c'|c]$ as the prediction score for "c entails c'" (i.e., the hypernym relation: c' is a hypernym of c). In particular, we use a family of classifiers $entails_t(c, c') = \mathbf{1}[\Pr[c'|c] > t]$, where $\mathbf{1}[\cdot]$ is the indicator function and t a threshold parameter. The conditional probability $\Pr[c'|c]$ is defined as the following. Denote e_c and $e_{c'}$ as the concept embeddings for c and c'.

- For the box embedding space, $\Pr[c'|c] = \Pr[e_{c'} \cap e_c] / \Pr[e_c]$, where $\Pr[e_{c'} \cap e_c]$ and $\Pr[e_c]$ have been defined in Section 7.2.1.

- For the hyperplane embedding space, $\Pr[c'|c] = \sigma(e_c^T e_{c'}/\tau - \gamma)$, where σ is the sigmoid function, $\tau = 0.125$, and $\gamma = 2d$ are the same set of scalar hyperparameters used for classifies objects. d is the embedding dimension.

- For the hypercone embedding space, $\Pr[c'|c] = \sigma((\langle e_{c'}, e_c \rangle - \gamma)/\tau)$, where $\langle \cdot, \cdot \rangle$ is the cosine similarity function, and $\tau = 0.1, \gamma = 0.2$ are the same set of scalar hyperparameters used for classifying objects.

By varying the threshold t, we compute the AUC-ROC score for FALCON-G based on three different embedding spaces, shown in Table G.11. This results suggest that the predicted concept embeddings from FALCON-G (Box) indeed better capture the hypernym relations between concepts. Note that the choice of specific hyperparameters: τ, γ, etc., will not affect the AUC-ROC score. We also want to highlight that the hyperplane and hypercone embedding spaces do not naturally support a conditional probability measure for two concepts, and thus are not optimized for capturing concept entailments, yielding a lower AUC-ROC score.

Methods	FALCON (Box)	FALCON (Hyperplane)	FALCON (Hypercone)
AUC-ROC Score	0.855	0.638	0.585

Table G.11: **AUC-ROC score of classifying entailment relationship based on concept embeddings from different embedding spaces.**

G.4 Ablation Details

This section discusses two ablation studies on how the number of base concepts in pretraining and the effect of the number of related concepts in supplemental sentences.

G.4.1 The Effect of the Number of Base Concepts

In this section, we will conduct an ablation study on how the number of base concepts can contribute to the model performance. We design a new split of the CUB dataset derived from 50 training species (130 base concepts), 50 validation species (81 concepts), and 100 test species (155 concepts). In our original experimental setup, we have used 100 training concepts (211 base concepts).

We perform the same fast concept learning experiments on this new split, using our model FALCON-G, a concept-centric baseline NSCL+GNN, and an end-to-end baseline MAC. All concept-centric models use box embeddings spaces.

Our results are summarized in Table G.12. Since we leverage the relationship between the novel concept and known concepts during the learning of the novel concept, all methods have an accuracy drop when there are fewer base concepts. This demonstrates that transfering knowledge from concepts already learned is helpful. Moreover, our model FALCON-G still has the highest accuracy when the number of base concepts is reduced.

# of base concepts	FALCON-G	NSCL+GNN	MAC
130	76.32	75.21	65.16
211	81.33	78.50	73.88

Table G.12: **Fast concept learning (with supplemental sentence) performance under different number of base concepts.**

% of all related concepts	FALCON-G	NSCL+GNN	MAC
0	76.37	73.38	73.55
25	80.20	77.16	73.94
50	80.78	76.93	74.13
75	81.20	77.77	74.23
100	81.33	78.50	73.88

Table G.13: **Fast concept learning performance under different percentage of related concepts in the supplemental sentence.**

G.4.2 The Effect of The Number of Related Concepts in Supplemental Sentences

In this section, we design an ablation study on how the number of supplemental sentences would affect model performance. Since all information captured in the supplemental sentences are the names of related concepts and their relations with the novel concept, we provide an ablation on the effect of the number of related concepts in supplemental sentences.

In the following experiments, we evaluate several models on the fast concept learning tasks on the CUB dataset. For each model, we use supplemental sentences containing 0% (no supp. sentence), 25%, 50%, 75%, and 100% (the setting described in the main paper) of all related concepts. Again, we compare our model FALCON-G, a concept-centric baseline NSCL+GNN, and an end-to-end baseline MAC. All concept-centric models use box embeddings spaces.

The results are summarized in Table G.13 and Fig. G-4. We have the following observations.

1. More related concepts included in the supplemental sentence generally lead to higher accuracy, across all models.

2. In both ceoncept-centric models (FALCON-G and NSCL+GNN), the most significant improvement in test accuracy occurs between 0% and 25%. This suggests that these models can benefit from even just an incomplete set of related concept information.

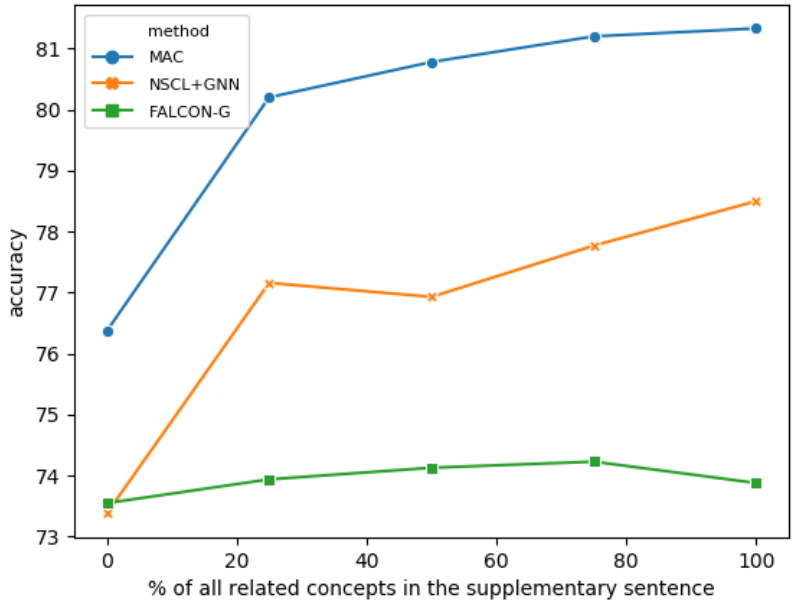


Figure G-4: **Fast concept learning performance under different percentages of related concepts in the supplemental sentence.**

3. Our model, FALCON-G, performs consistently the best across all percentages of related concepts.

G.5 Failure Mode Details

This section discusses how failure may occur in the detection module, the semantic parser, and the concept learning module, which may or may not result in a wrong answer.

G.5.1 Failure mode in detection module

There are a few circumstances where the detection module from the feature extractor produces an error. Here we list two cases where the detection module may fail, illustrated in Fig. G-5. In one scenario, the model may fail to detect one object from the scene, as in Fig. G-5a. Such errors usually occur among small and partially occluded objects. In the latter scenario, the model may recognize one object mask that does not correspond to any object, creating a false positive, as in Fig. G-5b. However, the example object may be referenced correctly, as the error does not affect the evaluation of the program in the neural symbolic execution phase. In all scenarios, the detection error leads to an incorrect object or a semantically ambiguous object being referenced during program evaluation. Such an

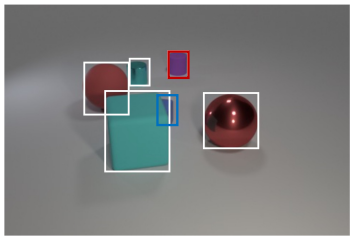
error may propagate to the meta-learning stage, change the novel concept's embedding, and eventually result in a correct or wrong answer.

G.5.2 Failure mode in semantic parser

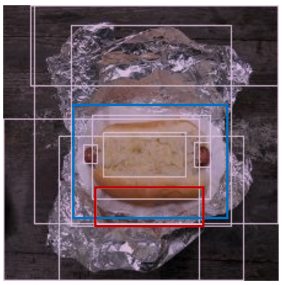
There are a few circumstances where the semantic parser produces an error. Here we list three cases where the semantic parser may fail, illustrated in Table G.14. First, the model may produce a synthetically wrong program. For example, there are invalid concept arguments in the output program. Following Johnson *et al.* [2017b], the parser applies heuristics to the output, including removing the invalid subprograms where concept arguments are illegal, and produces a feasible program, as in Table G.14a. In other circumstances, the model produces a syntactically correct but semantically wrong program. We will evaluate the program as it is produced by the Seq2seq. Such evaluation will either lead to the right answer or lead to the wrong answer. It should be no surprise that a semantically incorrect program may lead to a wrong answer since the concept arguments are wrong, as in Table G.14b. A semantically wrong program may also lead to the correct answer if the question has redundancy in concept quantifiers that refers to the correct object, as in Table G.14c. Currently, there is no mechanism implemented trying to recover from the semantic errors. In the future, we hope to incorporate methods such as REINFORCE (as in Yi *et al.* [2018] and NS-CL presented in Chapter 6) to leverage visual information to refine the semantic parser.

G.5.3 Failure mode in concept learning module

Besides errors in the semantic parser and the detection module, the concept learning module may also lead to potential errors due to imperfect visual groundings. Here we list two cases where the concept learning module may fail, illustrated in Fig. G-6. In the first scenario, the model encounters an error when filtering for a relational concept. The concept in the program is too extensive and filters out more candidates than desired. Such error results in a wrong object example, as in Fig. G-6a. In another scenario, the model encounters an error when filtering for a novel concept recently meta-learned. The concept in the program is too exclusive and filters out fewer candidates than desired. Such error results in a wrong answer in the final result, as in Fig. G-6b. In general, errors may take place in the concept learner when filtering for a base concept, a relational concept, or the novel concept just meta-learned when the visual grounding is imperfect. Such errors may result in different filter results with more or fewer objects being included. Further operation on the wrong filter output may eventually end up with a correct or incorrect answer.

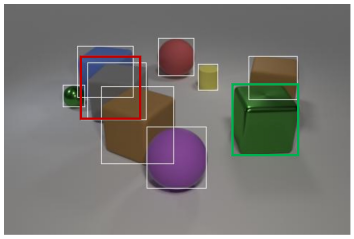


Descriptive sentence:The blue matte thing to the right of the big cyan object is a cube.
Expected example object: The blue cube around the upper-right corner of the cyan cube.
Predicted example object: The purple metal cylinder. ✗
Detection error: The blue cube is not detected by the detection module.

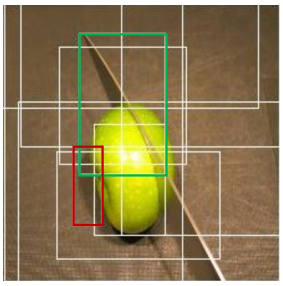


Descriptive sentence: The white small object is a plate.
Expected example object: The white plate.
Predicted example object: The lower side of the plate. ✗
Detection error: The entirety of the white plate is not detected as one object

(a) Error case when the detection module misses an object. The blue boxes indicates the expected object that the detection module misses; the red boxes indicates the wrongly predicted objects.



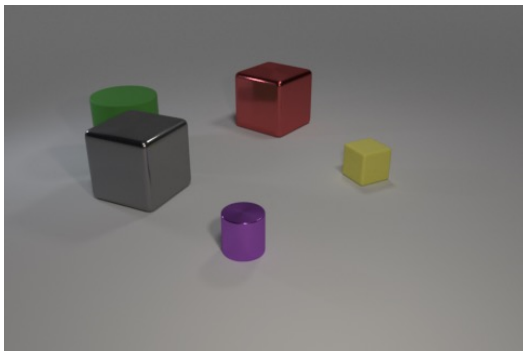
Descriptive sentence:The metallic thing that is right of the tiny sphere is a cube.
Expected example object: The green cube.
Predicted example object: The green cube. ✓
Detection error: The union of the blue cube and the gray cube is detected as one object by the detection module.



Descriptive sentence:The large long object is a silver object.
Expected example object: The long knife.
Predicted example object: The long knife. ✓
Detection error: Shadows of the knife and the apple are detected as objects by the detection module.

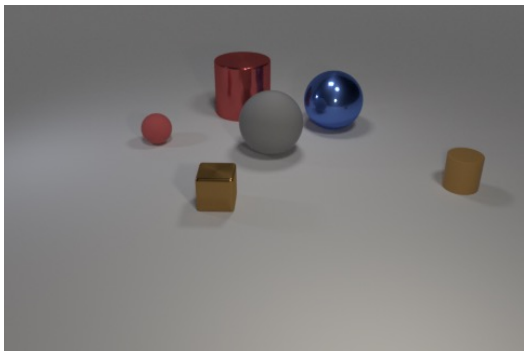
(b) Error case when the detection module recognize a false positive as an object. The green boxes indicates the expected and predicted object; the red boxes indicates false positives.

Figure G-5: **A visualization of three failure cases of the detection module.** The first two descriptive sentences are evaluated to find the wrong example object, and the last two are evaluated to find the right example object.



Descriptive sentence: The shiny block that are behind small rubber thing is a red object.
Expected example object: The metallic red cube.
Predicted example object: The metallic grey cube.
Explanation: The relational concept *behind* is not accurate enough; it is so extensive that even the grey cube is behind the yellow cube.

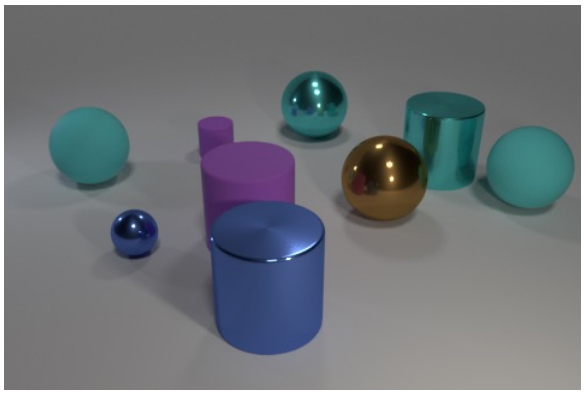
(a) Error case when the imperfect visual grounding leads to a wrong object example in the training image. This is part of the task that learns the novel concept *red*.



Question: How many red cylinders are to the right of the blue sphere?
Expected answer: 1. (The red cylinder)
Predicted answer: 0.
Explanation: The meta-learned concept *cylinder* is not accurate enough; it is too exclusive that even the red cylinder is not considered a *cylinder*.

(b) Error case when the imperfect visual grounding leads to a wrong answer in downstream question-answering. This is part of the task that learns the novel concept *cylinder*.

Figure G-6: **A visualization of two failure cases of the concept learning module.**



Text	Are there an equal number of small balls in front of the small ball and big cyan things?

Expected Program	`CountEqual(Filter(Filter(Relate(`
	`  Filter(Filter(Scene(), small), ball),`
	`  front), small), ball),`
	`  Filter(Filter(Scene(), big), cyan))`

Predicted Program	`CountEqual(Filter(Filter(Relate(`
	`  Filter(Filter(Scene(), small), ball),`
	`  front), small), Filter),`
	`  Filter(Filter(Scene(), big), cyan))`

Fixed Program	`CountEqual(Filter(Relate(`
	`  Filter(Filter(Scene(), small), ball),`
	`  front), small),`
	`  Filter(Filter(Scene(), big), cyan))`

(a) A syntactically infeasible program. The program is fixed with heuristics.

Text	Is there a metal cyan object to the left of the big brown object?

Expected Program	`Exists(Filter(Filter(Relate(`
	`  Filter(Filter(Scene(), big), brown),`
	`left), metal), cyan))`

Predicted Program	`Exists(Filter(Filter(Relate(`
	`  Filter(Filter(Scene(), big), purple),`
	`left), metal), cyan))`

Expected answer Yes. **Predicted answer** No.	

(b) A syntactically feasible but semantically wrong program, which yields a wrong answer.

Text	Is there a big cyan object to the right of the big brown object?

Expected Program	`Exists(Filter(Filter(Relate(`
	`  Filter(Filter(Scene(), big), brown),`
	`right), big), cyan))`

Predicted Program	`Exists(Filter(Filter(Relate(`
	`  Filter(Filter(Scene(), big), purple),`
	`right), big), cyan))`

Expected answer Yes. **Predicted answer** Yes.	

(c) A syntactically feasible but semantically wrong program, but the answer is correct.

Table G.14: **A visualization of three failure cases of the semantic parser.** All examples are from the question answering part of tasks on learning the novel concept *cyan* based on the image above.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix H

Appendix for Concept Learning Across Domains and Modalities

This chapter is organized as the following. In Appendix H.1, we formally define the first-order logic programs used by LEFT as well as discuss the broader impact of our method. In Appendix H.2, we present experiment results with error bars and details on the construction of zero-shot transfer tasks. In Appendix H.3, we provide implementation, training details, and code.

H.1 LEFT

H.1.1 Function definitions

We summarize the function definitions of LEFT in Table H.1. We note that $\mathcal{E}(expr)$ represents the execution result of *expr*. If the input to *concept* functions contains a concrete entity (e.g., result of *iota*), we take the dot product between the entity and the MLP result. If the argument is text (e.g., color), the function is implemented as a generative function instead of a discriminative one. Ternary and high-arity relations can be generalized similarly to unary and binary relations.

H.1.2 Broader impact

LEFT leverages a pre-trained large language model as its language interpreter, and hence, even though our prompts are general examples of first-order logic, we do not have direct control over the LLM's generation. The LLM may output harmful biases, which we will highlight as an important warning in our code release.

Signature & Implementation	Semantics
exists(var, expr) $\longrightarrow b$: *boolean* **Implementation:** $b = max(\mathcal{E}(expr))$	Return true if there is at least one assignment of *var* that satisfies *expr*.
forall(var, expr) $\longrightarrow b$: *boolean* **Implementation:** $b = min(\mathcal{E}(expr))$	Return true if there is all assignments to *var* satisfy *expr*.
iota(var, expr) $\longrightarrow e$: *entity* **Implementation:** $e = softmax(\mathcal{E}(expr))$	Return an assignment to *var*'s that satisfies *expr*.
not(expr) $\longrightarrow e$: *boolean* **Implementation:** $e_i = -\mathcal{E}(expr)_i$, for all $i \in \{1, 2, \cdots, N\}$	Compute the negation of an expression.
and($expr_1$, $expr_2$) $\longrightarrow e$: *boolean* **Implementation:** $e_i = min(\mathcal{E}(expr_1)_i, \mathcal{E}(expr_2)_i)$, for all $i \in \{1, 2, \cdots, N\}$	Compute the conjunction of two expressions.
or($expr_1$, $expr_2$) $\longrightarrow e$: *boolean* **Implementation:** $e_i = max(\mathcal{E}(expr_1)_i, \mathcal{E}(expr_2)_i)$, for all $i \in \{1, 2, \cdots, N\}$	Compute the disjunction of two expressions.
count(var, expr) $\longrightarrow c$: *integer* **Implementation:** $c = \sum_i [\sigma(\mathcal{E}(expr)_i)]$	Return the count of assignments to *var*'s that satisfies *expr*.
greater_than($expr_1$, $expr_2$) $\longrightarrow b$: *boolean* **Implementation:** $b = \sigma(\tau \cdot (\mathcal{E}(expr_1) - \mathcal{E}(expr_2) - \gamma))$	Return a score indicating whether $\mathcal{E}(expr_1)$ is greater than $\mathcal{E}(expr_2)$.
view(expr) **Implementation:** $\mathcal{E}(expr)$	Set the object returned by $\mathcal{E}(expr)$ as the object that the agent is looking at.
do(expr) **Implementation:** $\mathcal{E}(expr)$	Perform action computed by $\mathcal{E}(expr)$.
describe(var, expr) $\longrightarrow e$: *text* **Implementation:** $e = softmax(\mathcal{E}(expr))$	Return the best assignment to *var* (e.g., colors, shapes, etc.) that describes the object returned by $\mathcal{E}(expr)$.
concept(var_x) $\longrightarrow e$: *boolean* **Implementation:** $e_i = \text{MLP}^{\text{concept}}\left(f_i^{\text{unary}}\right)$	A unary function that takes objects, actions, and texts as inputs and returns Boolean values.
concept(var_x, var_y) $\longrightarrow e$: *boolean* **Implementation:** $e_{i,j} = \text{MLP}^{\text{concept}}(f_{i,j}^{\text{binary}})$	A binary function that takes objects, actions, and texts as inputs and returns Boolean values.

Table H.1: **Function definitions for first-order logic programs in LEFT.**

H.2 Experiments

H.2.1 Error bars

We present LEFT performance with error bars, taken from three runs with different seeds. The results for CLEVR [Johnson *et al.*, 2017a], CLEVR-10%, and CLEVR-Humans are below. All values are averaged over three random seeds, and the ± sign shows the computed standard error.

	CLEVR-10%	CLEVR-100%	CLEVR-Humans
LEFT	$0.942_{\pm 0.008}$	$0.996_{\pm 0.001}$	$0.780_{\pm 0.003}$ / $0.694_{\pm 0.001}$

Table H.2: **Results on the CLEVR dataset with standard errors.** Average performance and standard errors are computed based on three random seeds.

H.2.2 Transfer task construction

We detail the construction processes for CLEVR-Ref, CLEVR-Puzzles, and CLEVR-RPM below.

CLEVR-Ref We generate the CLEVR-Ref dataset using a subset of the templates from the CLEVR-Ref+ [Liu *et al.*, 2019b] dataset. We did not directly use the CLEVR-Ref+ dataset because it contains ordinal number concepts (in particular, the first, the second, etc.) that does appear in the original CLEVR dataset. Concretely, we used the following four templates:

1. {Select, Find, Point to} the *Object 1*.
2. {Select, Find, Point to} the *Object 1* (that is) *Relation 1* the *Object 2*.
3. {Select, Find, Point to} the *Object 1* (that is) *Relation 1* the *Object 2* and *Relation 2* the *Object 3*.
4. There is a(n) *Object 2* (that is) *Relation 2* the *Object 3*, {select, find, point to} the *Object 1* (that is) *Relation 1* it.

Here, items such as *Object 1*, *Object 2*, and *Object 3* will be replaced by object-level descriptors such as "shiny cube." *Relation 1* and *Relation 2* will be replaced by relational concepts in the dataset.

CLEVR-Puzzles The CLEVR-Puzzles dataset is generated based on the following template: "Can you find four objects from the image such that: *Object i* is *description*; *Object i* is *relation object j*." Here, *Object i*, *object j* will be replaced by "Object 1," "Object 2," "Object 3," or "Object 4;" descriptions will be replaced by object-level descriptions such as "shiny cube;" relations will be replaced by relational concepts in the dataset. In our data generation, we use four object descriptions (one for each object) and three object relation descriptions. Furthermore, we make sure that none of the object-level descriptions can uniquely identify an object, but the global solution is unique.

CLEVR-RPM The CLEVR-RPM dataset is generated based on the following template: "There are 9 objects, ordered in a 3x3 grid: row 1 col 1 is a *description 1*; row 1 col 2 is a *description 2*; row 1 col 3 is a *description 3*; row 2 col 1 is a *description 4*; row 2 col 2 is a *description 5*; row 2 col 3 is a *description 6*; row 3 col 1 is a *description 7*; row 3 col 2 is a *description 8*; I am missing one object at row 3 col 3. Can you find an object in the scene that can fit there?" Here, descriptions are object-level descriptions such as "shiny cube." Following existing work on Raven's Progressive Matrices [Barrett *et al.*, 2018], we randomly choose the "progression dimensions" from color, shape, material, and size for rows and columns.

H.3 Model

H.3.1 Implementation details

LEFT consists of several modular components in code. The first is a GPT-3.5 [Brown *et al.*, 2020] backed language interpreter, which takes as input queries from each dataset, and outputs first-order logic. The domain initializer takes the generated first-order logic programs and automatically initializes concept embeddings based on the parsed arity. The domain-specific grounding modules extract entity and relational features from the input modality. Finally, the domain-independent executor executes differentiable logic programs with the outputs of the grounding modules. We release our code below.

H.3.2 Training details

We use the official data splits released for each dataset, CLEVR [Johnson *et al.*, 2017a], ReferIt3D [Achlioptas *et al.*, 2020], HumanMotionQA [Endo *et al.*, 2023], and Cliport [Shridhar *et al.*, 2022a]. The core hyperparameters were set as 128 for concept embedding dimensions, and learning rate taken from prior neuro-symbolic concept learning repositories that we use as baselines. The full details are provided with the code, linked below.

Appendix I

Appendix for Visual Lexicon Learning

In this chapter, we describe the domain specific languages used in our experiments (Section I.1), demonstrate how the proposed CKY-E2 method works by a concrete example (Section I.2.1), show formal properties of CKY-E2 (Section I.2.2), present dataset setups and analyze model behaviors (Section I.3).

I.1 Domain Specific Languages and Neuro-Symbolic Reasoning

In this section, we will present and discuss the domain-specific languages (DSLs) we use for two domains: visual reasoning and language-guided navigation. We will further introduce the neuro-symbolic module we have designed for executing programs in these two domains. Overall, each DSL contains a set of types and a set of deterministic modules that have been manually designed for realizing necessary operations in these domains. However, in contrast to realizing them as we do in standard programming languages (with for-loops and if-conditions), we will be using tensor operations (e.g., tensor additions and multiplications) to realize them so that the output of each program is differentiable with respect to all of its inputs.

I.1.1 Visual Reasoning DSL

Our domain-specific language (DSL) for the visual reasoning domain is based on the CLEVR DSL introduced in [Johnson *et al.*, 2017a], and the neuro-symbolic realization of each functional module is slightly modified from the Neuro-Symbolic Concept Learner introduced in Chapter 6. We refer readers to the original papers for a detailed introduction to the DSL and neuro-symbolic program execution. Here we only highlight the key aspects

of our language and its neuro-symbolic realization, and discuss the difference between our implementation and the ones in original papers.

Our visual reasoning DSL is a subset of CLEVR, containing 6 types and 8 primitive operations. Table I.1 illustrates all 6 types and how they are internally represented in neuro-symbolic execution.

Type	Note	Representation
ObjConcept	Object-level concepts.	An embedding vector.
Attribute	Object-level attributes.	A vector of length K_{obj}, where K_{obj} is the number of
RelConcept	Relational concepts.	An embedding vector.
ObjectSet	A set of objects in the scene.	A vector $\mathbf{m}$ of length N, where N is the number of objects in the scene. Each entry $\mathbf{m}_i$ is a real value in $[0, 1]$, which can be interpreted as the probability that object i is in this set.
Integer	An integer.	A single non-negative real value, which can be interpreted as the "expected" value of this integer.
Bool	A Boolean value.	A single real value in $[0, 1]$, which can be interpreted as the probability that this Boolean value is true.

Table I.1: **The type system of the domain-specific language for visual reasoning.**

Table I.2 further shows all operations in the DSL. There are two main differences between the DSL used by G2L2 and the original CLEVR DSL. First, we have removed the *unique* operation, whose semantic meaning was to return the single object in a set of objects. For example, it can be used to represent the meaning of word "*the*" in "*the red object*", in which the semantic program of "*red object*" yields a set of red objects and the semantic program of "*the*" selects the unique object in that set. However, the meaning of "*the*" may have slightly different semantic type in different contexts, for example, "*what is the color of ...*". Since this has violated our assumption about each word having only one lexicon entry, we choose to remove this operation to simplify the learning problem. Meanwhile, to handle the "uniqueness" of the object being referred to, in our realization of related operations, such as *relate* and *query*, we will implicitly choose the unique object being referred to, which we will detail in the following paragraphs.

Signature	Note
scene() ⟶ ObjectSet	Return all objects in the scene.
filter(**a**: ObjectSet, *c*: ObjConcept) ⟶ ObjectSet	Filter out a set of objects having the object-level concept (e.g., red) from the input object set.
relate(**a**: ObjectSet, **b**: ObjectSet, *c*: RelConcept) ⟶ ObjectSet	Filter out a set of objects in set **a** that have the relational concept (e.g., left) with the input object **b**.
intersection(**a**: ObjectSet, **b**: ObjectSet) ⟶ ObjectSet	Return the intersection of set **a** and set **b**.
union(**a**: ObjectSet, **b**: ObjectSet) ⟶ ObjectSet	Return the union of set **a** and set **b**.
query(**a**: ObjectSet, *c*: Attribute) ⟶ ObjConcept	Query the attribute (e.g., color) of the input object **a**.
exist(**a**: ObjectSet) ⟶ Bool	Check if the set is empty.
count(**a**: ObjectSet) ⟶ Integer	Count the number of objects in the input set.

Table I.2: **All operations in the domain-specific language for visual reasoning.**

Object-centric scene representation. In our visual reasoning domain, we have assumed access to a pre-trained object-detector that generates a list of bounding boxes of objects in the scene. In our implementation, following Chapter 6, we use a pre-trained Mask R-CNN [He *et al.*, 2017] to generate bounding boxes for each object proposal. These bounding boxes, paired with the original image, are then sent to a ResNet-34 [He *et al.*, 2015] to extract a region-based representation (by RoI Align) and image-based representation, respectively. We concatenate them to form a vector embedding for each object in the image.

Neuro-symbolic realization. The high-level idea for the program execution is to build a collection of functions that realize the semantics of each operation based on the vector embeddings of objects and concepts. Taking the *filter* operation as an example, denote **a** as a vector representation of the input set, o_i the object embeddings, and e_c the concept embedding. We compute the vector representation **b** of the output set as:

$$\mathbf{b}_i = \mathbf{a_i} \cdot \sigma\left(\langle o_i, e_c \rangle\right),$$

where σ is the sigmoid function, and $\langle \cdot, \cdot, \rangle$ is the inner product of two vectors. Intuitively, we first compute the inner product between the concept embedding e_c and each object

Program	Type	Value
scene()	ObjectSet	[1, 1, 1]
filter(*scene*(), **CUBE**)	ObjectSet	[0.8, 0.1, 0.9]
filter(*filter*(*scene*(), **CUBE**), **SHINY**)	ObjectSet	[0.08, 0.08, 0.81] =[0.8, 0.1, 0.9] $\odot$ [0.1, 0.8, 0.9]
count(*filter*(*filter*(*scene*(), **CUBE**), **SHINY**))	Integer	0.97 = *sum*([0.08, 0.08, 0.81])

Table I.3: **An illustrative execution trace of the program *count*(*filter*(*filter*(*scene*(), CUBE), SHINY)).** *sum* denotes the "reduced sum" operation of a vector, which returns the summation of all entries in that vector. $\odot$ denotes element-wise multiplication for two vectors.

embedding, which gives as a vector of scores of whether object i has concept c. Next, we compute the element-wise multiplication between two vectors.

A key difference between our realization of these operations and the one in Chapter 6 is that we use element-wise multiplication to simulate the intersection between two sets, and $1 - (1 - \mathbf{a})(1 - \mathbf{b})$ for union. In contrast, Chapter 6 use element-wise min operation for intersection and max for union. Both realizations can be motivated by real-valued logic: product logic *vs*. Gödel logic. The main purpose of using products instead of min-max's is to make our realization compatible with our expected execution mechanism, which we will detail in Appendix I.2.

Example. Here we run a concrete example to illustrate the execution process of a program in the visual reasoning domain. Suppose we have an image containing three objects o_1, o_2 and o_3. We have two additional vector embeddings for concepts **SHINY** and **CUBE**. Furthermore, $\sigma(\langle o_i, e_{\textbf{SHINY}}\rangle) = [0.1, 0.8, 0.9]$, and $\sigma(\langle o_i, e_{\textbf{CUBE}}\rangle) = [0.8, 0.1, 0.9]$.

Consider the input sentence "*How many shiny cubes are there*". Table I.3 illustrates a step-by-step execution of the underlying program: *count*(*filter*(*filter*(*scene*(), **CUBE**), **SHINY**)).

Expected execution. In the visual reasoning domain, we have only implemented the expected execution mechanism for subordinate program trees whose type is *objset*, although many other types such as *integer* and *bool* also naturally supports expected execution. This is because, types such as *integer* and *bool* only appear at the sentence-level, and thus computing the "expectation" of such programs does not reduce the overall complexity.

Formally, the expected execution process compresses a list of semantic programs $v_1, v_2, \cdots, v_K$ and their corresponding weights $\tau(v_i)$ into a single semantic program v^* with weight $\tau(v^*)$. Suppose all v_i's have type *objset*. We use $\bar{v}_i$ to denote the execution result

of these programs. Each of them is a vector of length N, where N is the number of objects in the scene. We compute $\bar{v}^*$ and $\tau(v^*)$ as the following:

$$\begin{aligned} \bar{v}^* &= \frac{1}{\sum_i \exp(\tau(v_i))} \sum_i \left(\exp(\tau(v_i)) \cdot \bar{v}_i\right), \\ \tau(v^*) &= \log \sum_i \exp(\tau(v_i)). \end{aligned}$$

Intuitively, we normalize the weights using a softmax function to translate them into a distribution. Then we compute the expectation of the vectors. For more details about the definition and properties of expected execution, please refer to our main text and Appendix I.2.

Candidate lexicons. Recall that the process of lexicon learning has three stages. First, we generate an extensive collection of candidate semantic programs. Second, we generate candidate lexicon entries for each word by enumerating all possible candidate semantic programs generated in the first step and all possible ordering (linearization in a sentence) of its arguments. Third, we apply our CKY-E2 and gradient-based optimization to update the weights associated with each lexicon entry.

In our visual reasoning domain, we only consider the following candidate semantic programs and linearizations:

1. Syntactic type: *objset*, semantic program: *scene*() (English noun).
2. Syntactic type: *objset*, semantic program: *filter*(*scene*(), **?**) (English noun).
3. Syntactic type: *objset*/*objset*, semantic program: $\lambda x.$*filter*(x, **?**) (English adjective).
4. Syntactic type: *objset**objset*/*objset*, semantic program: $\lambda x.\lambda y.$*relate*(x, y, **?**) (English preposition I).
5. Syntactic type: *objset**objset*/*objset*, semantic program: $\lambda x.\lambda y.$*relate*(y, x, **?**) (English preposition II)
6. Syntactic type: *bool*/*objset*, semantic program: $\lambda x.$*exist*(x).
7. Syntactic type: *integer*/*objset*, $\lambda x.$*count*(x).
8. Syntactic type: *word*/*objset*, $\lambda x.$*query*(x, **?**).
9. Syntactic type: CONJ_{AND}, $\lambda f.\lambda g.(\lambda x.$*intersect*($f(x), g(x)$)) (generalized conjunction).
10. Syntactic type: CONJ_{OR}, $\lambda x.\lambda y.(\lambda z.$*intersect*($z$, *union*($x, y$))) (generalized disjunction).

As we will see later, when we compare the candidate lexicon entries for the visual reasoning domain and the language-driven navigation domain, the visual reasoning domain contains significantly fewer entries than the navigation domain. This is because much of the learning process in this domain is associated with learning the concept embeddings. In the following few paragraphs, we will explain how we instantiate concepts based on these lexicon entry templates and implement generalized conjunction and disjunction in our domain.

First, for each word (more precisely, word type), e.g., *shiny*, we will instantiate 10 lexicon entries. For semantic programs that contain unbounded concept arguments (**?** marks), we will introduce a series word-type-specific concepts. Specifically in this domain, each word type will be associated with 3 concept representations: $\textbf{SHINY}_{\text{ObjConcept}}$, $\textbf{SHINY}_{\text{RelConcept}}$, and $\textbf{SHINY}_{\text{Attribute}}$. Based on Table I.2, the first two concepts will be represented as two embedding vectors, and the the third concept will be represented as a vector, indicating which concepts belong to this attribute category. Next, we will instantiate these lexicon entries by filling in these concept representations. For example, one of the candidate lexicon entry for *shiny* is syntactic type: *objset*, semantic program: $\textit{filter}(\textit{scene}(), \textbf{SHINY}_{\text{ObjConcept}})$. During training, all these vector embeddings as well as the weights associated with each lexicon entry, will be optimized jointly.

Next, we discuss the implementation for two conjunctive lexicon entries. The grammar rule for CONJ_{AND} is:

$$T \;\; \text{CONJ}_{\text{AND}} \;\; T \rightarrow T,$$

where T is an arbitrary syntactic type (thus called generalized conjunction). There are two typical use cases: what is the shape of the red and shiny object, and what is the shape of the object that is left of the cube and right of the sphere. In the first case, both arguments have syntactic type *objset*/*objset*. In the second case, both arguments have syntactic type *objset**objset*. Note that CLEVR contains only the second case.

The grammar rule for CONJ_{OR} is:

$$\textit{objset} \;\; \text{CONJ}_{\text{OR}} \;\; \textit{objset} \rightarrow \textit{objset}\backslash\textit{objset}.$$

It covers the case: *how many objects are blue cubes or red spheres.* Our implementation is slightly different with human-defined lexicon entries for the word *or*, in particular, because the DSL we use is a small set of set-theoretic operations, which does not fully match the expressiveness of truth-conditional semantics. Thus, the current DSL does not support the representation of all words in the dataset (in particular, *or* and *are*). Thus, we have implemented this ad-hoc fix to handle disjunction.

Finally, we want to emphasize again that, since our DSL does not support representing

Word Type	Syntactic Type	Semantic Program
are	¡EMPTY¿	¡EMPTY¿
there	¡EMPTY¿	¡EMPTY¿
any	*bool*/*objset*	$\lambda x.exist(x)$
shiny	*objset*/*objset*	$\lambda x.filter(x, \textbf{SHINY}_{\text{ObjConcept}})$
cubes	*objset*	$filter(scene(), \textbf{CUBE}_{\text{ObjConcept}})$

Table I.4: **The learned lexicon entries associated with each word for a simple sentence:** *are there any shiny cubes?***.** The derived semantic program for the full sentence is $exist(filter(filter(scene(), \textbf{CUBE}_{\text{ObjConcept}}), \textbf{SHINY}_{\text{ObjConcept}}))$

Signature	Note
empty() $\longrightarrow$ ActSeq	Create an empty string (of length 0).
newprim() $\longrightarrow$ ActSeq	Create a string containing only one primitive action. In SCAN, there are in total 6 primitives.
newint() $\longrightarrow$ Integer	Create a single integer. In SCAN, we only support integers {2, 3, 4}.
concat(**a**: ActSeq, *c*: ActSeq) $\longrightarrow$ ActSeq	Concatenate two input strings.
repeat(**a**: ActSeq, **b**: Integer) $\longrightarrow$ ActSeq	Repeat the input string for multiple times.

Table I.5: **All operations in the domain-specific language for language-driven navigation.**

all semantic programs of words, we allow certain words to be associated with an "*empty*" lexicon entry. This entry can be combined with any words or constituents next to it and does not participate in the composition of syntactic types and semantic programs. In Table I.4 we show the lexicon entry associated with each word in the sentence "*are there any shiny cubes?*", learned by our model, G2L2.

I.1.2 Language-Driven Navigation DSL

Our DSL for the language-driven navigation domain is a simple string manipulation language that supports creating new strings, concatenating two strings, and repeating a string multiple times. Our DSL contains only two primitive types: action sequence, abbreviated as ActSeq, and integer.

Formally, we summarize the list of operations in our language-driven navigation domain in Table I.5.

Probabilistic string representation. We represent each string in a "probabilistic" manner. In particular, each string s is represented as a tuple $\langle L^s, C^s \rangle$. L^s is a categorical distribution of the length. C^s is a three-dimensional tensor, indexed by ℓ, k, c, where $C^s_{\ell,k,c} = p(s[k] = c | length(s) = \ell)$. Thus, C has the shape $[L+1, L, |V|]$, where L is the max length of a string and V is the action vocabulary. For simplicity, we constrain that $C^s_{\ell,k,c} \equiv 0$ for all $k > \ell$.

It is straightforward to represent empty strings: $L_0 = 1$, or strings with a single action primitive a: $L_1 = 1$ and $C_{1,0,a} = 1$. Now we explain our implementation of the *concat* and the *repeat* operation.

For $z = concat(x, y)$:

$$\begin{aligned} L^z_\ell &= \sum_{0 \le i \le \ell} \left(L^x_i \cdot L^y_{(\ell - i)} \right); \\ C^z_{\ell,k,c} &= \frac{1}{L^z_\ell} \sum_{0 \le i \le \ell} \left(L^x_i \cdot L^y_{(\ell-i)} \cdot (C^x_{i,k,c} + C^y_{\ell-i,k-i,c}) \right). \end{aligned}$$

The high-level idea is to enumerate the possible length of both strings.

Similarly, for $z = repeat(x, m)$,

$$\begin{aligned} L^z_\ell &= \begin{cases} L^x_{\ell/m} & \text{if } \ell \bmod m = 0 \\ 0 & \text{otherwise} \end{cases} \\ C^z_{\ell,k,c} &= \begin{cases} L^x_{\ell/m, k \bmod (\ell/m), c} & \text{if } \ell \mod m = 0 \text{ and } k < \ell \\ 0 & \text{otherwise} \end{cases}. \end{aligned}$$

Expected execution. In the language-driven navigation domain, we only perform expected execution for semantic programs of type ActSeq, whose execution results can be represented using the probabilistic string representation. Denote $\bar{s}_i$ as the execution results for K programs, and $\tau(s_i)$ the corresponding weights. We define $p(s_i) = softmax(\{\tau(s_i)\})_i = \frac{\exp \tau(s_i)}{\sum_j \exp \tau(s) j)}$. We compute the expected string $\bar{s}$ and its weight $\tau(s)$ as:

$$L^s_\ell = \sum_i p(s_i) L^{s_i}_\ell; \quad C^s_{\ell,k,c} = \frac{\sum_i \left(p(s_i) L^{s_i}_\ell \cdot C^{s_i}_{\ell,k,c} \right)}{L^s_\ell}.$$

Candidate lexicons. We use a simple enumerate algorithm to generate candidate lexicon entries for our language-driven navigation DSL. Specifically, we first enumerate candidate semantic programs for each lexicon entry that satisfy the following constraints:

1. There are at most three operations.
2. There are at most two arguments.

Type	Program (Note)
ActSeq	*walk*() The simplest program that constructs a string with a single action primitive: **WALK**.
(ActSeq) ⟶ ActSeq	$\lambda x.concat(look(), x)$ Prepend a **LOOK** action to an input string.
(ActSeq, ActSeq) ⟶ ActSeq	$\lambda x.\lambda y.concat(x, y)$ Concatenate two strings.
(ActSeq, ActSeq) ⟶ ActSeq	$\lambda x.\lambda y.concat(repeat(x, 2), y)$ Repeat the first string twice and concatenate with the second string.
((ActSeq) → ActSeq, ActSeq) ⟶ ActSeq	$\lambda x.\lambda y.concat(y, x(walk()))$ The first argument (x) is a function which maps a ActSeq to another ActSeq. The second argument y is an ActSeq. The function invokes x with a simple string **WALK**, and concatenate the result with y.

Table I.6: **Sample semantic programs generated by the enumeration process based on our language-driven navigation DSL.**

3. There is at most one argument whose type is a functor.

4. There is no argument of type *Integer*.

Table I.6 lists a couple of programs generated by the algorithm and their corresponding types.

Based on the candidate semantic types, we first instantiate candidate lexicon entries by enumerating possible ordering (linearization) of the arguments. For example, the simple program $\lambda x.concat(look(), x)$ has two possible linearizations: *ActSeq/ActSeq* and *ActSeq\ActSeq*. As discussed in the main paper, in order to handle parsing ambiguities, we further introduce two finer-grained syntactic types for the *ActSeq* type: *S* and *V*. In practice, we only allow the following set of syntactic types: *V*, *V/V*, *V\V*, *V\V/V*, *V\V/(V\V)*, and *S\V/V*. In total, we have 178 candidate lexicon entries for each word.

Word Type	Syntactic Type	Semantic Program	exp(Weight)
ONE	N	1	1.0
PLUS_ONE	N\N	$\lambda x.x + 1$	0.5
PLUS_ONE	N\N	$\lambda x.x \times 3$	0.5
MUL_THREE	N\N	$\lambda x.x + 1$	0.5
MUL_THREE	N\N	$\lambda x.x \times 3$	0.5

Table I.7: **A set of candidate lexicon entries and their weights in a simple arithmetic domain.**

Index	Syn. Type	Semantic Program ($derivation_k$)	Execution Result ($exec_k$)	exp(Weight)
1	N	$(1 + 1) + 1$	3	$0.25 = 1 \times 0.5 \times 0.5$
2	N	$(1 \times 3) + 1$	4	$0.25 = 1 \times 0.5 \times 0.5$
3	N	$(1 + 1) \times 3$	6	$0.25 = 1 \times 0.5 \times 0.5$
4	N	$(1 \times 3) \times 3$	9	$0.25 = 1 \times 0.5 \times 0.5$

Table I.8: **Four candidate derivations of the simple sentence "*ONE PLUS_ONE MUL_THREE*" in a simple arithmetic domain.**

I.2 Delve Into Expected Execution

In this section, we will run a concrete example in a small arithmetic domain to demonstrate the idea of expected execution. Following that, we will prove an important invariance property that has guided our realization of different functional modules in both domains.

I.2.1 CKY-E2 In An Arithmetic Domain

In this section, we will consider parsing a very simple sentence in an arithmetic domain. We will be using numbers and two arithmetic operations: + and ×. Each number in the domain will be represented as a real value.

Suppose we have the following lexicon entries associated with three words, illustrated in Table I.7. There are 4 candidate derivations of the sentence "*ONE PLUS_ONE MUL_THREE*", as illustrated in Table I.8. For simplicity, we will show the exp of weights. Thus, the probability of a derivation is proportional to the product of all lexicon entry weights.

Suppose that we will be using the groundtruth execution result of this program as the supervision, applied by an L2 loss. Then we will be interested in the expected execution

result of all possible derivations. In this case, it is

$$\mathbb{E}[exec] = 0.25 \times 3 + 0.25 \times 4 + 0.25 \times 6 + 0.25 \times 9 = 5.5.$$

Next, we will try to accelerate the computation of $\mathbb{E}[exec]$ by doing local marginalization. Consider the constituent "*ONE PLUS_ONE*". In CKY-E2, this will be the first constituent that the algorithm constructs. It has two possible derivations, whose corresponding semantic programs are $(1 + 1)$ and (1×3). Both derivations have the same syntactic type N, and thus, they will be combined with N\N on its right, in the next step. In this case, in CKY-E2, we will merge these two derivations into one (again, since we only care about the expected execution result, not the set of all possible derivations!). The combined derivation has value $0.5 \times (1 + 1) + 0.5 \times (1 \times 3) = 2.5$, and total weight $0.5 + 0.5 = 1$.

Then, when we are trying to compose the derivation for the whole sentence, i.e., combine the constituent "*ONE PLUS_ONE*" and "*MUL_THREE*", we no longer need to compute all $2 \times 2 = 4$ possible derivations, but only $1 \times 2 = 2$ derivations. They are: $2.5 + 1 = 3.5$, with probability 0.5, and $2.5 \times 3 = 7.5$, with probability 0.5. In this case, we see that taking local marginalization reduces the computation complexity of parsing and retains the expected execution result!

I.2.2 Formal Properties of CKY-E2

Motivated by the intuitive example shown above, now let us formally specify the properties of CKY-E2.

Expectation invariance. Consider the composition of two consecutive constituents a and b. We use $a_1, \cdots a_N$ and $b_1, \cdots b_N$ to denote possible derivations of both constituents. We assume all a_i's are of the same syntactic type without loss of generality, so do all b_i's, since we will handle different syntactic types separately.

Denote f as the semantic composition function for a and b. Without local marginalization, we will have in total $N \times M$ derivations for the result constituent: $c_{i,j} = f(a_i, b_j)$. We further use $\bar{a}_i$, $\bar{b}_j$, and $\bar{c}_{i,j}$ to denote the execution results of these derivations. Without derivations, the expected execution results is:

$$\begin{aligned}
\mathbb{E}[\bar{c}] &= \frac{1}{\sum_{i,j} \exp \tau(c_{i,j})} \sum_{i,j} \left(\exp \tau(c_{i,j}) \cdot \bar{c}_{i,j} \right) \\
&= \frac{1}{\sum_{i,j} \exp \tau(a_i) \cdot \exp \tau(b_j)} \sum_{i,j} \left(\exp \tau(a_i) \cdot \exp \tau(b_j) \cdot f(\bar{a}_i, \bar{b}_j) \right).
\end{aligned}$$

Again, without loss of generality, we will assume $\sum_i \exp \tau(a_i) = 1$ and $\sum_j \exp \tau(b_j) = 1$, because any constant scaling of these weights will not change the expectation $\mathbb{E}(\bar{c})$. Thus, we simplify this definition as,

$$\mathbb{E}[\bar{c}] = \sum_{i,j} \left(\exp \tau(a_i) \cdot \exp \tau(b_j) \cdot f(\bar{a}_i, \bar{b}_j) \right).$$

Let us assume function f has the following property: $\mathbb{E}[f(a, b)] = f(\mathbb{E}(a), \mathbb{E}(b))$, which expands as,

$$\sum_{i,j} \left(\exp \tau(a_i) \cdot \exp \tau(b_j) f(\bar{a}_i, \bar{b}_j) \right) = f\left(\sum_i \left(\exp \tau(a_i) \bar{a}_i \right), \sum_j \left(\exp \tau(b_j) \bar{b}_j \right) \right).$$

Thus, locally marginalizing the expected value for $\mathbb{E}[\bar{a}]$ and $\mathbb{E}[\bar{b}]$ will not change the expected execution result of c.

In this simple proof we have made a strong assumption on the composition function f: $\mathbb{E}[f(a, b)] = f(\mathbb{E}(a), \mathbb{E}(b))$. In practice, this is true when f are addition or multiplication functions between scalars, vectors, matrices, and in general, tensors. It will not apply to element-wise min/max operations and other non-linear transformations. Fortunately, this already covers most of the operations we use in the visual reasoning and language-driven navigation DSLs. In practice, even if some operations do not have this property, we can still use this mechanism to approximate the expected execution result.

Although we have only proved this property for binary functions, the idea itself easily generalizes to unary functions, such as the negation operation, and higher-arity functions. Furthermore, by induction over derivation trees, we can easily prove that, as long as all composition functions satisfy the expectation invariance property, applying CKY-E2 yields the same result as doing the marginalization at a sentence level.

Complexity. In general, it is hard to quantify the reduction in computational complexity by doing local marginalization. However, we can still estimate the number of possible derivations constructed in the entire CKY-E2 procedure. For simplicity, consider the case where there is only one primitive syntactic type: X. Moreover, there are N_0 candidate lexicon entries of type X; N_1 entries of type X/X, and N_2 entries of type $X\backslash X/X$. For each span, considered in the CKY-E2 algorithm, all possible derivations associated with this span can be grouped into 4 categories:

1. Derivations of type X. In this case, only 1 derivation will be retained (merged by the expected execution result).

2. Derivations of type X/X. In this case, they must be a primitive lexicon entry. Thus,

there are at most N_1 of them.

3. Derivations of type $X\backslash X/X$. Similarly, at most N_2 of them.

4. Derivations of type $X\backslash X$. This intermediate syntactic type is a result of a partial composition between $X\backslash X/X$ and X (on its right). Thus, there are at most N_2 of them.

Overall, there are at most $1 + N_1 + 2 \times N_2$ derivations stored for this span. Since the total span is $O(L^2)$, where L is the total length of the input sentence, the overall complexity of CKY-E2 is a polynomial of L, N_0, N_1, and N_2, which is significantly lower than an exponential number of derivations.

I.2.3 Connection with Other Parsing Models

Connection with synchronous grammars. Our approach can be viewed as defining a synchronous grammar over joint (parse tree, meaning program) pairings. We didn't use this framing in the main paper because typical applications of synchronous grammar involve parallel datasets (e.g., sentence pairs in two languages for machine translation, or sentence–image pairs for generating image descriptions) in which the information in both modalities is directly parsed. In our setting, in contrast, the meaning-program component of the synchronous grammar is acquired through more distant supervision. We will make all this clear in the final version, including stating how the chart parsing process can be seen as synchronously constructing parsing trees and meaning programs. The expected execution can be viewed as a "compression" step over all meaning programs that can be potentially parsed from each span.

Connection with sum-product CKY. There are also connections between CKY-E2 and sum-product CKY, such as the shared Markovian assumption, but would like to add that the main difference between CKY-E2 and sum-product CKY is that CKY-E2 computes only the "expectation" of the execution results of the underlying program. Instead, sum-product CKY computes a full distribution of the parsing results (e.g., in syntax parsing, it can compute the categorical distribution of the root symbol). Sum-Product CKY can not be applied to our setting, because we are dealing with programs and the space of possible programs is infinite. Modeling a distribution of all possible programs might be intractable; instead, computing the expectation is much easier, and this enables us to do local marginalization.

I.3 Experimental Setup And Analysis

In this section, we will present in detail the experimental setups for both datasets: CLEVR and SCAN. Both datasets are released under a BSD license. Specifically, we will include details about the setups for different compositional generalization tests. Although some of them (the ones in the SCAN dataset) have already been illustrated in their original paper, we echo them here for completeness. Next, we will also analyze the performance of each model on both datasets, focusing on the inductive biases they have in their model and how these inductive biases contribute to their compositional generalization in different splits.

I.3.1 Visual Reasoning: CLEVR

We start with the dataset generation process for the CLEVR dataset. Next, we formally present the dataset generation protocol for all splits. Furthermore, we analyze the performance of various models.

Baselines. As a quick recap of our baselines, MAC [Hudson and Manning, 2018] uses an end-to-end vision-language attention mechanism to process the question and image jointly; TbD-Nets [Mascharka *et al.*, 2018] and NS-VQA [Yi *et al.*, 2018] uses a neural sequence-to-sequence model (with attention, see [Bahdanau *et al.*, 2015] for semantic parsing. The parser is trained with sentence-program pairs; NS-CL from Chapter 6 uses a customized sequence-to-tree model, and jointly learns the visual recognition models and the semantic parser. One crucial implementation detail with the semantic parser module in NS-CL is that, it uses additional token embeddings to annotate the concepts appearing in the question. When generating a concept token in the semantic program, it uses an attention mechanism to select the concept from the input question.

Dataset generation. Since we only consider the cases where each word is associated with a unique lexicon entry, we have manually excluded sentences that will break this assumption. Among all of the 425 templates in the original CLEVR dataset [Johnson *et al.*, 2017a], we have retained 208 templates. Specifically, we have removed all templates that involve: 1) coreference resolution, 2) "same"-related questions, and 3) number comparison-related questions. To keep the number of questions the same as the original dataset, we choose to re-generate the questions using the selected subset of templates, following the original data generation protocol. All our splits are generated based on this basic version, which we name as the standard training set and the standard test set.

Split: data efficiency. We test the data efficiency of models by only using 10% of training data in the standard training set, and test the models on the standard test set.

In this split, the semantic parsers used by all program-based methods: TbD-Nets, NS-VQA, and NS-CL, have nearly perfect accuracy. Thus, the performance drops are primarily due to the limited data for training individual modules. Overall, TbD-Nets have the worst data efficiency. There is no performance drop for the NS-VQA model, because the visual recognition modules are pretrained with direct object-level supervision.

Split: compositional generalization (purple). The training set is generated by selecting all questions that either do not contain the word "purple" or have a length smaller than or equal to 7 (including punctuation). The test set is generated by selecting all sentences containing the word "purple" and has a length greater than 7.

In this split, the semantic parsers used by all program-based methods: TbD-Nets, NS-VQA, and NS-CL have nearly perfect accuracy. Thus, the performance drops are primarily due to 1) the limited data for training individual modules (in this case, *filter*(*purple*) and 2) novel composition of learned modules. Overall, NS-VQA and NS-CL answer more questions correctly than TbD-Nets.

Split: compositional generalization (right). The training set is generated by selecting all questions that either do not contain the word "right" or have a length smaller than or equal to 12. The test set is generated by selecting all sentences that contain the word "right" and have a length greater than 12.

In this split, the semantic parser of NS-CL still yields almost perfect accuracy. In contrast, the accuracy of the neural sequence-to-sequence parser used by TbD-Nets and NS-VQA is around 91%. Thus, the performance drop of TbD-Nets is mainly due to the inferior performance of the corresponding neural module: *relate*(*right*). Compared with the realization of a *filter* module (used in our *purple* generalization test), the *relate* module in TbD-Nets has a significantly deeper neural architecture (6 layers vs. 3 layer). Thus, we hypothesize that this module requires more data to train.

Split: compositional generalization (count). The training set is generated by selecting all questions that either do not contain operation "count" or have a length smaller than or equal to 9. The test set is generated by selecting all sentences that contain operation "count" and have a length greater than 9.

Among all compositional generalization tests, this is the most challenging one. The semantic parser, in this case, need to generalize from: "*how many cubes are there*" and "*what's the shape of the object that is both left of the cube and right of the sphere?*", to "*how*

many cubes are both left of the cube and right of the sphere?" We have constructed the training data in a way such that all constituents have been seen in the training data. In this test, the program-level accuracy of the semantic parser used by TbD-Nets and NS-VQA is 70.8%. NS-VQA outputs slightly higher QA accuracy.

We find that, for the parsing module in NS-CL, it fails to output the correct filter operation. Given the input question "what is the number of spheres that are right of the cube", it sometimes outputs *filter_cube relate_right filter_cube count* (the correct program has the third operation *filter_sphere* instead). This is because the system has never seen sentences composed of "counting" operations and such complex structures; during training, it has only seen short sentences such as "what is the number of spheres?" Only our G2L2 model, with its explicit constituent-level compositionality, is able to make these generalizations.

Split: depth generalization. We define the "hop number" of a question as the number of intermediate objects being referred to in order to locate the target object. For example, the "hop number" of the question "*how many red objects are right of the cube?*" is 1. We train different models on 0-hop and 1-hop questions and test them on 2-hop questions.

This generalization test evaluates the generalization to deeper syntactic structures. All methods except for our model G2L2 fail on this test. By evaluating the accuracy of the program generated by different semantic parsers, we find that, the neural sequence-to-sequence model used by TbD-Nets and NS-VQA completely fails on this task, sometimes generating invalid programs (the program-level accuracy is 1.7%). Thus, we see a significant performance drop for both methods. Meanwhile, NS-CL also generates wrong programs, but the programs are always valid due to its sequence-to-tree design. Furthermore, even if the program is not correct, for example, they miss certain operations, the execution result may still lead to a correct answer. For example, as long as the semantic parser gets the outer-most filter operation (i.e., the last hop) correct, it is still possible to generate the correct answer.

I.3.2 Language-Driven Navigation: SCAN

The SCAN dataset [Lake and Baroni, 2018] consists of several official splits for generalizability evaluation. Following existing work, we evaluate on the splits corresponding to the generalization test of "jump" and "around right". In addition, we test the generalizability across different output lengths and the data efficiency of models. The performance is measured by exact match–based output accuracy.

Split: data efficiency. The official "simple" split randomly samples 80% among all possible example pairs as the training set, and leaves the others as the test set. We use the

training set of the official simple split (available at [this url]) as the entire training set, and test the data efficiency by using only 10% of them. We sample the 10% data uniformly for each input length. Both settings are tested in the official simple test split, which is available at [this url].

Split: compositional generalization (jump). The training split consists of *jump* in isolation, i.e., the input is *jump* while the ground-truth output is I_JUMP, along with all other examples that do not contain *jump*. The model is expected to recognize that *jump* has the same syntactic category as other verbs such as *run*, and does well on complicated instructions including *jump*, e.g., mapping *jump twice* to I_JUMP I_JUMP. The training data is available at [this url] and the test data is available at [this url].

Split: compositional generalization (around right). Similar to the "jump" test, the training set for the "around-right" test consists of all possible examples that do not contain *around right* in their inputs, while the test set consists of all examples that have *around right*. It is worth noting that different from *jump*, *around right* in isolation is not a valid input as it lacks a primitive. The model is expected to perform compositional generalization, understanding *around right* based on existing training phrases such as *around left* and *opposite right*. The training data is available at [this url] and the test data is available at [this url].

Split: length generalization. The model is expected to perform well on examples with long ground-truth output while training those with short ground-truth output. In this test, all training examples consist of less than or equal to 22 tokens in their outputs, while the output of a test example may consist of up to 48 tokens. The training data is available at [this url] and the test data is available at [this url].

Baseline models. All baseline models are built on top of a seq2seq model [Sutskever *et al.*, 2014]: the original seq2seq model [Sutskever *et al.*, 2014] trains an LSTM-based encoder-decoder model using the training set; the other methods augment the training set by either heuristics or learned models, and train an LSTM-based encoder-decoder model using the augmented data. It is worth noting that among all considered baseline methods, GECA [Andreas, 2020] may generate examples in the test set, especially for compositional generalization tests since the heuristics it introduces is by nature compositional.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix J

Appendix for Manipulation Mechanisms

J.1 Method Detail

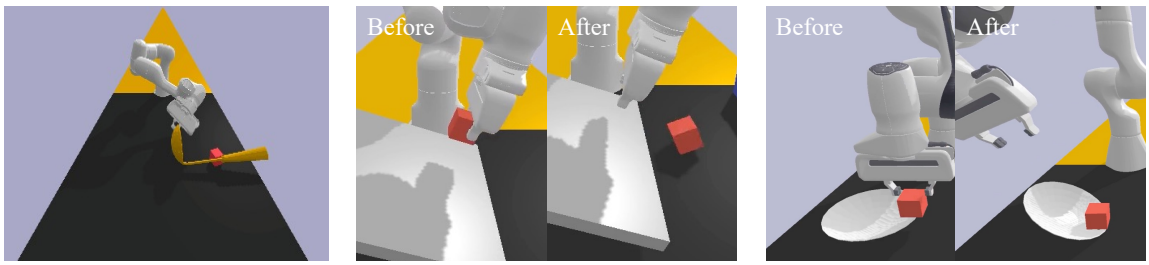
J.1.1 Planning Algorithm

Algorithm 13 shows the bi-level search algorithm we use. In the discrete search level, we enumerate both basis operations as well as mechanism operations. During the continuous parameter search phase, for basis operations instantiated from mechanisms, we use the mechanism-specific sampler rather than the generic sampler for continuous parameters.

Algorithm 13 Bilevel Search Algorithm

```
1:  procedure BilevelSearch(s_0, g, operator_schemas)
2:    plan_gen ← SymbolicSearch(s_0, g, operator_schemas)        ▷ Explore discrete plans
3:    for all plan ∈ plan_gen do          ▷ For all candidate sequences of basis operations
4:      ContinuousSearch(s_0, g, plan)
5:  procedure ContinuousSearch(s_0, g, plan)
6:    grounded_plan ← ∅;  s ← s_0
7:    for all op ∈ plan do
8:      for all arg ∈ op.args do
9:        arg ← InvokeSampler(op.sampler)          ▷ Generate continuous parameters
10:     if CheckPrecondition(op, s) then
11:       s ← T(s, op)                  ▷ Simulate the operator with sampled parameters.
12:       grounded_plan ← grounded_plan ∪ {op}
13:     else break
14:   if IsGoalAchieved(grounded_plan, g) then
15:     return grounded_plan            ▷ Return the first plan that achieves the goal
16:   return empty
```

J.1.2 Briefly Dynamic Manipulation



Case 1: non-rigid attachments between objects while moving.

Case 2: object pose change after placement due to physics.

Case 3: support object pose change after new objects being placed.

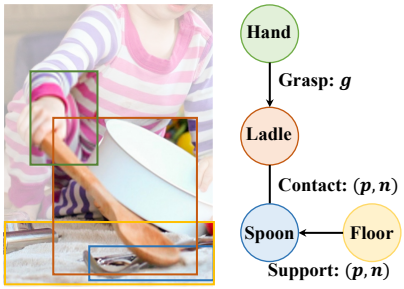
Figure J-1: **Illustration of three briefly-dynamic manipulation scenarios in the paper.**

The system can handle robot-object and object-object contact without assuming quasi-static motion. For example, when placing objects on surfaces, we consider subsequent pose changes: objects placed on inclined surfaces may slide down, and heavy objects placed on levers can alter the pose of the lever. Formally, we assume a *briefly-dynamic* manipulation setting, where the robot controller is position control-based, and manipulated objects may experience acceleration and velocity until they reach a stable configuration. Figure J-1 illustrates a few examples of briefly-dynamic manipulation tasks handled by our sampler and planner.

J.1.3 Basis Operations

In our manipulation context, each schema represents either a robot action that does not change the contact mode graph (e.g., moving the arm without object collisions) or a primitive action that modifies the contact mode (e.g., grasping an object). Table 10.1 shows the complete list of basis operators used in this paper.

The precondition and the effect of an action schema describe the contact relationships between objects before and after the execution. Fig. J-2 showcases a concrete example. The action schema involves three objects: the object being held *?tool*, the target object that is in contact with the tool object *?target*, and the object that is currently supporting the target object *?support*. Additionally, there are two continuous parameters: *?param* specifies the contact between *?target* and *?tool*, including the contact surface and contact normal; *?qt* specifies the robot arm trajectory as a sequence of joint-space waypoints. This action updates the robot joint angles, and the poses of *?tool* and *?target*. Given the discrete and continuous parameters, we use a joint-space position controller to execute the actions and use the execution results to update the state variables.




```
(:action move-with-contact
 :parameters (?tool - item ?target - item ?support - item
              ?param - contact-move-param ?qt - trajectory)
 :precondition (and
   (holding ?tool)
   (support ?target ?support)
   (valid-contact-move-param      ... ?param)      ;; points, normal
   (valid-contact-move-trajectory ... ?param ?qt)) ;; robot trajectory
 :effect (and
   (assign (robot-config) ...)   ;; update robot position
   (assign (pose ?tool)   ...)   ;; update tool position
   (assign (pose ?target) ...)  ;; update target position
```

Figure J-2: **The modeling of the robot basis operation *move-with-contact* using a STRIPS-like syntax.**

We will present the implementation details for the samplers associated with each operator in the next section (Appendix J.1.4). At a high level, these samplers are designed to be very generic: for grasping, it randomly samples two parallel surfaces on objects; for object-object contact, it randomly samples two surfaces on the object and then transforms the object held by the robot so that two surface normals point to each other.

J.1.4 Samplers for Generic Operations

Recall that there are three types of continuous variables to be sampled for the basis operators described in Table 10.1: grasping poses relative to an object (represented as $SE(3)$ poses of the end-effector relative to the object), placement poses (represented as $SE(3)$ poses in the support object frame), contacts between two objects (represented as the $SE(3)$ pose of object 1 in the frame of object 2), and robot arm trajectories (represented as a sequence of arm trajectories). Here, we supplement the list of samplers we use to generate these continuous parameters. They are designed to be generic, relying solely on geometry and not specific object semantics (e.g., soup ladle grasping).

Grasp (G). The grasp sampler, $G(O, T_o)$, accepts the object's shape and current pose, O and T_o respectively, and identifies two "parallel" surfaces on the object mesh, represented as (p_1, n_1) and (p_2, n_2), where p_1 and p_2 are two points and n_1 and n_2 are surface normals. The definition of being parallel is that: $(p_1 - p_2) \cdot n_1 = 1$ and $n_1 \cdot n_2 = -1$. It then computes a corresponding robot end-effector pose T_{ee} such that $T_e e$ centered at the midpoint between p_1 and p_2, and T_{ee} is perpendicular to n_1. It then checks the distance between two surfaces so that the parallel gripper can hold the object at T_{ee}. Finally, it checks the reachability of T_{ee} using an inverse-kinematics solver.

Placement (P). The placement position sampler, $P(O_1, O_2, T_{o2})$, considers the shapes of both the holding object, O_1, and the target support object, O_2, and the pose of O_2. It randomly samples two surfaces, represented as (p_1, n_1) and (p_2, n_2), one on each object such that $n_2 \cdot (0, 0, 1)^T > 0.9$ (i.e., n_2 is close to the $+z$ direction). Next, it solves for a transform

T on O_1 such that $Tp_1 = T_{o2}p_2$ and $Tn_1 = -T_{o2}n_2$ (essentially place p_1 on O_1 at p_2 and pointing towards n_2).

Object Contact (C). For both robot-object and object-object contact, the object contact sampler, $C(O_1, O_2, T_{o2}, O_s, T_s)$ takes five arguments, including the current holding object O_1 (or the robot gripper itself when not holding anything), the object to contact O_2 and its current pose T_{o2}, and the object that supports O_2: O_s and its pose T_s. It first randomly samples two surfaces, represented as (p_1, n_1) and (p_2, n_2) on O_1 and O_2 respectively. Since we do not consider pushing O_2 "towards" the supporting object O_s, we additionally require that n_2 is perpendicular to n_s, which is the direction of the support force from O_s to O_2. Next, it solves for a transform T on O_1 such that $Tp_1 = T_{o2}p_2$ and $Tn_1 = -T_{o2}n_2$ (essentially place p_1 on O_1 at p_2 and pointing towards n_2 to excert force).

Trajectory (T). For grasping and placement trajectories, the trajectory sampler, $T(T_{init}, T_{target})$, considers the initial and target end-effector pose of the robot gripper. It first uses an inverse kinematic solver to solve for two robot configurations at the designated end-effector pose q_{init} and q_{target}. Next, we compute a collision-free trajectory (except for collisions with the object being held and the object to contact) using a Bidirectional Rapidly-exploring Random Tree (BiRRT) algorithm.

For move-with-contact trajectories, the trajectory sampler, $T(T_{\text{init}}, p_1, n_1, p_2, n_2)$, accepts the initial configuration of the robot, g_{init}, and the contact surfaces on the two objects sampled using the object contact sampler C: (p_1, n_1) and (p_2, n_2). It proceeds to randomly sample a "push" distance, d, along the contact normal direction, n_1, from a uniform distribution in the range $[0.05, 0.25]$ meters. Subsequently, the sampler generates the arm trajectory by invoking the BiRRT algorithm to follow a set of waypoints corresponding to a linear Cartesian-space motion along n_1 by distance d.

J.2 Experiment Detail

J.2.1 Mechanism Learning Setup

Our evaluation encompasses six distinct mechanisms, grouped into two categories: the first four tasks assess "tool-use."

(Edge) pushing objects to the edge of a table for pickup. There are four object models used in this mechanism: plate, calculator, caliper, and document.

(Hook) using tools to reach for distant objects. There are five objects that can be used as the "hook:" wooden L-shape stick, soup ladle, hammer, spoon, and caliper.

(Lever) flipping objects using heavy objects as levers. There are four "heavy" objects that can be used to flip the plate: box, spoon, dipper, and walnut.

(Poking) using tools to poke objects out of a tunnel. There are three object models that can be used as the "poking" tool: wooden stick, spatula, and spoon.

The remaining two tasks fall under the "reasoning about stability" category.

(Center-of-Mass) achieving stable object placement on another object. There are three object models to be placed on the small block: plate, calculator, and document.

(Slope-and-Blocker) using objects as blockers to prevent objects from falling off inclined surfaces. There are three object models that can be used as the blocker: wooden stick, wooden L-shape stick, and spoon.

For each environment, we first manually defined a canonical pose for each object such that the mechanism is feasible. Next, for each training and testing instance, we randomly apply small translations (a uniform distribution within ± 5 centimeters) and small rotations (uniform within ± 15 degrees) to the canonical pose of each movable object.

J.2.2 Sampler Learning

Taking a closer look at the importance of sampler learning, Fig. J-3 illustrates a breakdown of the number of samples required for the "hook use" mechanism using our planning algorithm, with the generic sampler and with the learned sampler. Fig. 10-3b shows the inferred macro definition for this mechanism, and here we count the number of samples produced by each individual sampler. In this case, most of the samplers are produced to generate candidate grasping poses of the tool and possible contacts between the tool and the target (i.e., how to reach the tool).

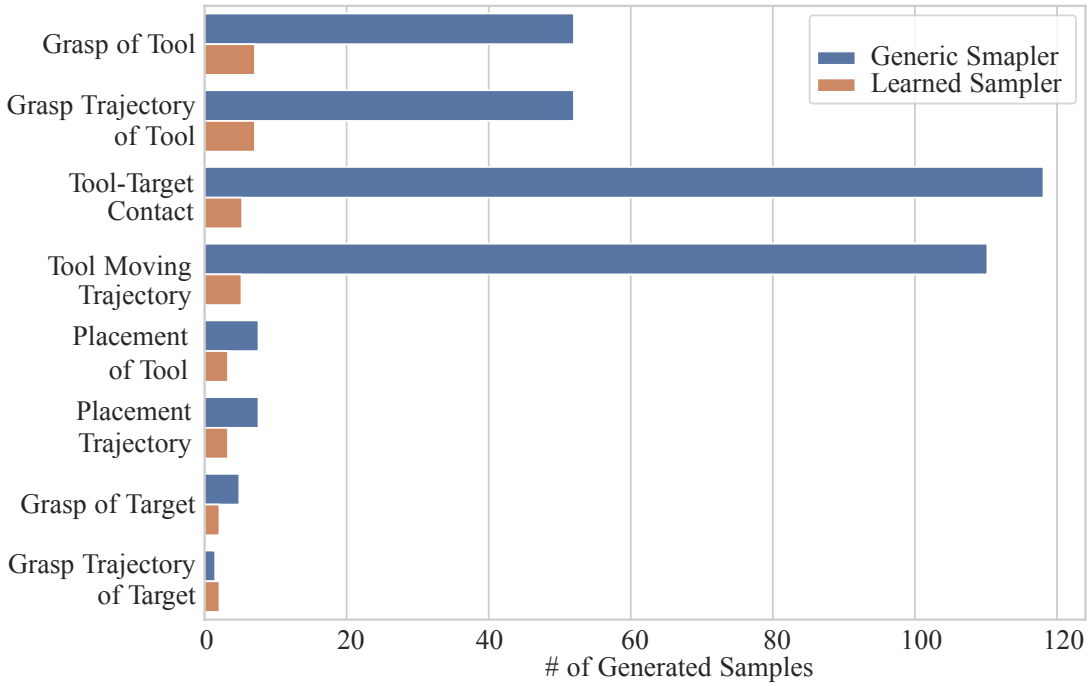


Figure J-3: **Breakdown of samples produced by different samplers for the hook-using task.**

J.2.3 Physical Robot Deployment

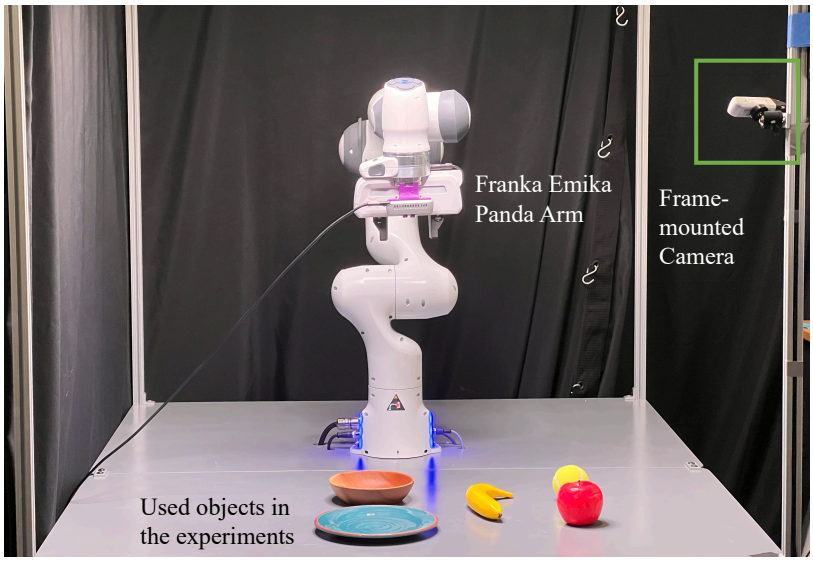


Figure J-4: **Our real-world setup.**

Task	push-to-edge(plate)	hook-use(banana, {ball, apple})
Succ. Rate	80%	70%

Table J.1: **Success rate of our physical robot setup on two representative mechanisms.**

Our real-world setup, shown in Fig. J-4, contains a Franka Emika Panda robot arm with a parallel gripper, mounted on a table. We also have an Intel Realsense D435 camera mounted on the table frame, pointing 45 degrees down. Our vision pipeline contains six steps: first, based on the calibrated camera intrinsics and extrinsics, we reconstruct a partial point cloud for the scene. Second, we crop the scene to exclude volumes outside the table (e.g., background drops, etc.) Third, we use a RANSAC-based algorithm to estimate the table plane, and extract all object point clouds on top of the table. Fourth, we use Mask-RCNN to detect all objects in the RGB space, and extract the corresponding point clouds. For objects that are not detected by the MaskRCNN, we first use DBScan to cluster their point clouds, and then run the SegmentAnything model to extract their segmentations in 2D, and subsequently the point clouds. Finally, we perform object completion by projecting and extruding the bottom surface for all detected objects down to the table.

Next, we import the reconstructed object models into our physical simulator PyBullet, and directly deploy our planning algorithm with the learned mechanisms to compute plans. Since we use the same robot model in simulation, we can directly execute the planned robot

trajectories in the real world. In practice, we execute in a closed-loop manner. If the grasp fails (which can be detected by the gripper sensor), we move the arm to its neutral position and replan. After each placement action and pushing action, we use the vision pipeline to obtain an updated world state and replan.

We demonstrate and evaluate the system on two example tasks: pushing plates to the edge and hook-using for distant objects. We use the same set of objects on the table but with different initial configurations. We repeat each experiment 10 times and report the success rate. Table J.1 summarizes the result. The visualization of our vision pipeline and the robot videos can be found on our website.

The push-to-edge task is relatively easier given the learned mechanism. The only two failure cases in our experiment were triggered by the robot pushing the plate too far and the plate falling off the table. The hook-using task is more challenging. There are two main failure modes we observe in the experiments. First, the planner favors grasping objects (e.g., the banana "hook") at the tip, which is a very unstable grasp—sometimes causing the executor to fail. Also, sometimes the planner generates wrong object motion for the objects because 3D of reconstruction errors: the planner is using a "hallucinated" part of the object to push the distant object—currently, we cannot recover from this type of failure. Finally, sometimes the sampled grasp is invalid, again because of the errors in object perception. In this case, our close-loop execution improves the performance.

Appendix K

Appendix for PDSketch

The chapter is organized as the following. First, in Section K.1, we provide additional results on data efficiency and on-policy learning for the BabyAI environment studied in the paper. Then, in Section K.2, we present an example-based formal introduction of the PDSketch language. Next, in Section K.3, we describe the implementation of our domain-independent heuristic. We also discuss why sparsity and locality structures are particularly important in inducing domain-independent heuristics. Finally, in Section K.4, we present our experiment setups. Both Section K.3 and Section K.4 are presented based on the language definition in Section K.2.

K.1 Additional BabyAI Results

In this section, we supplement two additional results on the BabyAI environment. First, in Section K.1.1, we present the data efficiency study of different models in terms of their performance-time success rate. We also make variance analysis of different models across different runs. Next, in Section K.1.2, we present how our model can be integrated with on-policy learning methods to make active data gathering.

K.1.1 Data Efficiency and Variance Analysis

We further quantify the data efficiency of different models in terms of their performance-time success rate. Note that this is different from the Figure 7 in the main paper, where we have studied the data efficiency in terms of model learning accuracy. In this section, we show that, 1) compared to model-free baselines, integrate model learning and planning does improve the overall data efficiency in the domain we consider, and 2) the same is true for different PDSketch models with different levels of abstractions and prior knowledge.

Concretely, we generate three datasets, with 100 demonstration episodes, 1000 episodes,

Model	100 episodes	1000 episodes	10000 episodes
BC	0.24 ± 0.01	0.24 ± 0.01	0.39 ± 0.03
DT	0.23 ± 0.02	0.24 ± 0.01	0.30 ± 0.05
PDS-Base	0.04 ± 0.00	0.11 ± 0.01	0.80 ± 0.07
PDS-Abs	0.04 ± 0.02	0.98 ± 0.03	**0.99** ± 0.01
PDS-Rob	**0.72** ± 0.09	**0.99** ± 0.01	**0.99** ± 0.01

Table K.1: **The planning success rate for different models using different amount of training data.**

and 10,000 episodes, respectively. We train the model on different datasets until converge, and evaluate their performance. We have repeated all experiments three times. The reported value are the average score and the standard deviation.

Results are summarized in Table K.1. Here we summarize our findings as the following.

1. First, in general, behavior cloning and decision transformer achieve similar performance on this task. When the amount of training data is small, the model perform noticeably worse than model-based methods.

2. The base model, even if trained with a large amount of data, struggle to capture the robotic movement transitions when facing obstacles. This results in that the performance gets stuck at around 80%.

3. Compared to recognizing object properties, a significant amount of data is required to learn the transition model. When the transition model is given (in PDS-Rob), learning the visual recognition models in this simple grid-world domain is very data-efficient: using only 100 episodes, the model successfully solves 70% of the tasks.

K.1.2 On-Policy Learning

Our system can also be integrated into an on-policy learning setting. This is compatible with the standard model-based reinforcement learning setting. Specifically, recall that in an interactive learning setting, the system receives the current state and a goal expression. The agent itself should decide the next action to take. In the PDSketch case, we run the planner based on the current model parameters θ to generate a plan. We follow the plan in the simulator and collect the resulting trajectory as well as the "done" signals from the environment. Finally, we update the model parameter θ using the newly collected data. Although our model may leverage offline data, to keep the algorithm simple, we do not implement data reusing. Within 10,000 episodes, our model PDS-Rob is able to reach 0.99

performance-time success rate. Comparatively, the reported performance of Proximal Policy Optimization (PPO) [Schulman *et al.*, 2017; Chevalier-Boisvert *et al.*, 2019] needs 200,000 episodes.

We found that direct application of the on-policy algorithm on models with less inductive biases (i.e., PDS-Abs and PDS-Base) does not yield successful results. The success rate stuck around 6%. This is primarily because of their failure in exploration: direct application of the planner generates successful trajectories at a very low probability. This can be potentially alleviated by adding random exploration factors or replanning when the agent fails to reach the goal.

Note that, since our model solely focuses on representing and learning the model, it is possible to integrate our framework with other model-based reinforcement learning algorithms, such as joint model and policy learning [Schrittwieser *et al.*, 2020], or world-model-based reinforcement learning [Hafner *et al.*, 2021]. We leave these extensions as a future work.

K.2 PDSketch Language

In this section, we detail the design, the syntax, the semantics, and the implementation of the PDSketch definition language. We present it in two parts: the predicate and action definition (Appendix K.2.1), and PDSketch expressions (Appendix K.2.2).

K.2.1 Predicate and Action Definition

PDSketch is based on the Planning-Domain Definition Language [Fikes and Nilsson, 1971; Fox and Long, 2003], which is a language specialized and derived from LISP. We choose a LISP-style language for two important reasons. First, compared to other procedure- or object-centric languages such as Python, LISP features an easy definition of first-order logic, especially for recursively defined rules. Second, a significant amount of planning domains have been written in PDDL. Extending the language reduces the learning cost for programmers and also make old PDDL definitions easily reusable in PDSketch.

For readers who are familiar with PDDL, a PDSketch file is a domain definition file (in contrast to a problem definition file). PDSketch only contains the definition of predicates and actions in the domain. It does not contain problem-specific information, such as the current state of the world, or the goal specification.

Thus, a PDSketch file contains three parts. A PDSketch definition, at the highest level, takes the following form:

```
definition: "(" "define" "domain" domain-name-def content-def* ")"
domain-name-def: "(" "domain" string ")"
content-def: type-def | predicate-def | derived-def | action-def
```

An empty domain file can be written as:

```
(define domain
  (domain my-domain-name)
)
```

This defines a domain with name “my-domain-name,” with no predicates and actions defined.

A `content-def` can be either a type definition, a predicate definition, or an action schema definition. First, type definition:

```
type-def:        "(" ":types" single-type-def ")"
single-type-def: type-name-list - base-type
type-name-list:  type-name+
type-name:       string
base-type:       type-name | prim-type | vector-type
prim-type:       "bool" | "int64" | "float32"
vector-type:     "vector" "[" prim-type (, int)? "]"
```

Below, we will be using the BabyAI world as an example, the example type definition of the BabyAI domain is:

```
(:types
  robot item - object
  pose       - vector[float32, 2]
  direction  - vector[int64, 1]
)
```

Specifically, a type definition contains multiple lines, each line being a pair of a type name list, and a base type. There are two types of types in PDSketch: object type and value type. Intuitively, object type refers to a concrete object in the world, while a value type usually denotes the return type of a feature function over the objects. Currently, we do not support the full hierarchical structure of type definitions: each type either inherits "object", which indicates this is an object type, or inherits a primitive value type (Boolean, integer, or floating-point numbers), or a vector type. In a vector-type definition, the second argument (e.g., the `2` in `vector[float32, 2]`) denotes the dimension of the vector. This can be omitted. Overall, the definition above defines four types: an object type called "robot," an object type called "item," a value type that is a 2D vector, named "pose," and a value type that is a 1D vector of integer (i.e., a single integer), named "direction."

The `predicate-def` and `derived-def` are jointly used to define predicates. A predicate can be either a predicate directly observable from the environment, or a "derived" predicate, which is a predicate whose value is computed based on the value of other predicates. We first show the grammar definition.

```
predicate-def:        "(" ":predicates" single-predicate-def ")"
single-predicate-def: "(" predicate-name kwargs? variable-list ")"
predicate-name:       string
kwargs:               "[" kwarg "]"
kwarg:                kwarg-key "=" kwarg-value
kwarg-value:          string | int | bool | float | base-type
variable-list:        typed-variable*
typed-variable:       variable-name "-" type-name
variable-name:        string
```

An example definition of some input predicates is the following:

```
(:predicates
  (robot-pose      [return_type=pose]            ?r - robot)
  (robot-direction [return_type=direction]       ?r - robot)
  (item-pose       [return_type=pose]            ?o - item)
  (item-image      [return_type=vector[float32]] ?o - item)
)
```

In this case, we have defined four predicates: a 2D position of the robot, the direction that the robot is facing, the 2D position of the item, and finally, an image representing a local crop of the item.

Based on the input predicates, users can define many "derived" predicates, whose values are automatically computed based on other input and derived predicates. The formal syntax is:

```
derived-def: "(" ":derived" single-predicate-def expr ")"
```

in which `expr` is a syntax for expressions, which we will detail later. As an example, we look at the definition of several item property-related predicates.

```
(:derived (item-feature [return_type=vector[float32, 64]] ?o - item)
  (??f (item-image ?o))
)
```

This definition defines a predicate named `item-feature`. It takes an item as its argument, and returns a 64-dimensional vector embedding associated with the object. Its expression is `(??f (item-image ?o))`. That is, an unknown mapping from the input image of the item to a vector embedding. This definition will implicitly define a function whose name is `derived::item-feature::f`. This function has no default implementation. The programmer, after loading the PDSketch definition, should register the corresponding implementation of this function. For example, one option is to associate this function with a convolutional neural network (CNN) that takes the image crop of each object `?o` and computes the corresponding item feature. Note that, in this definition, we are also implicitly defining a *parameter sharing* strategy: for all objects in the environment (independent of its absolute index), we will be applying the same CNN identically to them. We will detail possible expressions PDSketch supports later.

Based on this `item-feature` definition, users can define several classifiers that will be useful in defining goal predicates.

```
(:derived (is-red  ?o - item) (??f (item-feature ?o)))
```

```
(:derived (is-blue ?o - item) (??f (item-feature ?o)))
; ... more colors
(:derived (is-ball ?o - item) (??f (item-feature ?o)))
(:derived (is-door ?o - item) (??f (item-feature ?o)))
; ... more shapes
(:derived (is-open ?o - item) (??f (item-feature ?o)))  ; for doors
```

By default, when the return type is not specified, the function returns Boolean values, which is consistent with the original PDDL syntax.

In BabyAI, items that the agent is holding has a special 2D position which is `[-1, -1]`. Thus, whether the agent is holding an object can be classified by

```
(:derived (robot-holding ?r - robot ?o - item)
  (??f (item-pose ?o))
)
```

Finally, we want to define a function indicating whether the robot is facing the object,

```
(:derived (robot-facing ?r - robot ?o - item)
  (??f (robot-pose ?r) (item-pose ?o))
)
```

In its expression, we are defining an unspecified function `??f`, which takes two arguments, the robot position, and the item position. With all these predicates, we can now define the tasks. Recall that the ActionObjDoor environment has the following three tasks:

```
(exists (?o - item) (and
  (robot-facing agent ?o) (is-red ?o) (is-box ?o)
))
```

go to a red box:

```
(exists (?o - item) (and
  (robot-holding agent ?o) (is-red ?o) (is-box ?o)
))
```

pick up a red box:

```
(exists (?o - item) (and
  (is-red ?o) (is-door ?o) (is-open ?o)
))
```

open a red door: In the first two goal specifications, we are using a name constant "agent" to denote the robot that we are considering.

Next, we are going to define actions. There are four actions the agent can perform. The syntax for action definition is:

```
action-def: "(" ":action" (action-name kwargs)
  ":parameter" "(" variable-list ")"
  ":precondition" expr
  ":effect" expr
")"
```

An action definition is composed of a name, a list of parameters, a precondition, and an effect. The parameter are the arguments to the action. The precondition is a Boolean output value, indicating the situation where this action can be executed. Note that, the precondition is not part of the transition model of the environment. Instead, it's a property that is associated with the underlying policy of this action. We will come back to this point when we discuss concrete examples. The effect is an expression that change the state variables according to certain rules.

Concretely, we start with a simple action: "lturn," which models the behavior that the robot turns left.

```
(:action lturn
 :parameters (?r - robot)
 :precondition (and )
 :effect (assign (robot-direction ?r) (??f (robot-direction ?r)))
 ; could also be written as:
 ; :effect (robot-direction::assign ?r (??f (robot-direction ?r)))
)
```

This definition has defined an action named "lturn," it takes a single parameter "?r" which indicates the robot that we want to control. It has an "empty" precondition, indicating that this primitive action can be executed anytime. Meanwhile, the effect is to change the "robot-direction" state variable associated with the robot `?r`. The rule is a based on an (unspecified) function that takes the current robot facing direction and returns the new direction: `(??f (robot-direction ?r))`.

With first-order logic formula, one can define more complex actions. One good example is the definition of the "forward" action. In BabyAI, the forward action takes the following rule. When there is no object facing the agent, the agent can move forward. When there is an object facing the robot, the robot may be blocked by the object, in which case the robot position will not change. Note that not all map items will block robots' movement. Which type of objects will block the robot is to be learned by the learning algorithm. In this case, we can define the robot action in an intuitive way as the following:

```
(:action forward
 :parameters (?r - robot)
 :precondition (and )
 :effect (robot-pose::assign ?r (??f
   (robot-pose ?r)
   (robot-direction ?r)
   (foreach (?o - item)          ; enumerate all items in the map
     (when (robot-facing ?r ?o)  ; if the robot is facing this item
       (item-feature ?o)         ; consider the item-feature of ?o
     )
   )
 ))
```

```
)
```

We now break down this definition into pieces. First, this is an action that takes only one argument (the robot "?r"). Similar to the case of "lturn," this action does not have a precondition, indicating that the action can be applied in any situations. The effect definition for this action is slightly more complex. First, the action changes the state variable `(robot-pose ?r)`. The new value is computed based on three things, the current `robot-pose`, the current `robot-direction`, and the feature of all objects `?o` that the robot is facing. Here, we are using two special keywords, `foreach` and `when`. The first one iterates over all items in the environment, and the second one selects the feature of object `?o` if the condition is satisfied. Importantly, in this case, the corresponding function `f` should be implemented as a variable-length-input function. That is, depending on how many objects are selected, the number of input features will change. In our implementation, this function is implemented as a graph neural network (GNN). Of course, if the programmer knows more about the underlying transition model, they can specify more detailed structure, such as the following:

```
; define a helper derived predicate.
(:derived (is-obstacle ?o - item) (??f (item-feature ?o)))
(:action forward-detail
 :parameters (?r - robot)
 :precondition (and )
 :effect (when
   ; when there is no item ?o such that
   ;   ?o is an obstacle and the robot is facing ?o
   (not (exists (?o - item) (and (is-obstacle ?o)
                                 (robot-facing ?r ?o)) ))
   ; the robot pose will move forward and the new pose
   ;   will be computed by the current pose and the facing direction.
   (assign (robot-pose ?r)
     (??f (robot-pose ?r) (robot-direction ?r))
   )
 )
)
```

Similarly, we can have the following definition for "the robot executes picking up."

```
; define a helper predicate indicating whether the object
;   can be picked up.
(:derived (can-pickup ?o) (??f (item-feature ?o)))
(:action pickup
 :parameters (?r - robot)
 :precondition (and )
 :effect (foreach (?o - item)
   (when (and (robot-facing ?r ?o) (can-pickup ?o) )
     (assign (item-pose ?o)
       ; a dummy function, that should be implemented to return
       ;   [-1, -1], indicating the item is in robot's inventory.
       (??f )
     )
   )
 )
)
```

Overall, the definition language allows users to encode various kinds of prior knowledge about the transition model into the representation. Meanwhile, it does not require human programmers to fully specify the details, especially the recognition and classification problems. The learning algorithm will follow the structure and learns the missing pieces.

Importantly, depend on the knowledge that the programmer has about the environment, such definition can be very detailed, or very abstract. At an extreme case, we can have the following definition:

```
(:action mysterious-action
 :parameter (?r - robot)
 :precondition (and )
 :effect (and
   (robot-pose::assign ?r (??f1
     (robot-pose ?r) (robot-direction ?r)
     (item-pose ??) (item-feature ??)
```

```
   ))
   (robot-direction::assign ?r (??f2
     (robot-pose ?r) (robot-direction ?r)
     (item-pose ??) (item-feature ??)
   ))
   (foreach (?o - item) (item-pose::assign ?o (??f3
     (robot-pose ?r) (robot-direction ?r)
     (item-pose ??) (item-feature ??)
   )))
   (foreach (?o - item) (item-feature::assign ?o (??f4
     (robot-pose ?r) (robot-direction ?r)
     (item-pose ??) (item-feature ??)
   )))
 )
)
```

Here we are using a syntax sugar: the notation `(item-pose ??)` is equivalent to `(foreach (?x - item) (item-pose ?x))`. That is, this function takes the pose of all items into consideration. Note that, in this case, we have *no* structure built in to the system: the action can change any state variable, and the change depends on all other variables in the state representation. In this case, depending on the actual implementations of `f1` to `f4`, this transition model can be implemented in any form, such as a graph neural network (thus an object-centric transition model).

Remark: Preconditions *vs*. conditional effects. There are two seemingly similar ways to define the condition under which an action takes effect: preconditions and conditional effects. To better understand the different between these two options, consider the following two examples:

```
(:predicates
  (p ?o - item)
  (q ?o - item)
)
(:action op1
 :parameters (?o - item)
```

```
 :precondition (p ?o)
 :effect (q:assign (??f (q ?o)))
)
(:action op2
 :parameters (?o - item)
 :precondition (and )
 :effect (when
   (p ?o)
   (q:assign (??f (q ?o)))
 )
)
```

The key difference is that, a "precondition" is the property associated with the corresponding low-level controller for the action, that is $\pi_{\texttt{op1}}$. Its semantics is that, the $\pi_{\texttt{op1}}$ can only be executed when `(p ?o)` returns true. It does not specify what will happen when we execute $\pi_{\texttt{op1}}$ in such a situation. In contrast, the semantics of the second definition is that, the corresponding $\pi_{\texttt{op2}}$ can be executed under any circumstances. However, only if the condition for the conditional effect is true (i.e., `(p ?o)`, the state value for `q` will be changed.

Thus, preconditions and conditional effects should be learned from different signals. Preconditions should be learned from the signal from the execution of the corresponding controller. More specifically, the controller $\pi_{\texttt{op1}}$ should return a Boolean value indicating whether the controller can be executed. By contrast, the conditional effects should be learned from the outcome of the execution of the controller $\pi_{\texttt{op2}}$.

Note that, the conditional effect formulation is the one that fits most of the concurrent environment setups for reinforcement learning. That is, the defined actions correspond to the primitive actions that the robot can execute. Thus, usually all actions are always applicable at any state (i.e., the precondition of all actions are empty).

K.2.2 Expressions

PDSketch has the following built-in operations.

Propositional logic operations: `and`, `or`, `not`, and `implies`. The propositional logic operations take the following syntax:

```
and-expr-def:     "(" "and"     expr* ")"
or-expr-def:      "(" "or"      expr* ")"
not-expr-def:     "(" "not"     expr  ")"
implies-expr-def: "(" "implies" expr expr ")"
```

The semantics of these operators follows the convention of Boolean operations: conjunction, disjunction, negation, and implication. Their implementation are based on the differentiable Gödel t-norms: $\texttt{not}(p) = 1 - p$, $\texttt{and}(p_1, p_2) = \min(p_1, p_2)$, $\texttt{or}(p_1, p_2) = \max(p_1, p_2)$, and $\texttt{implies}(p_1, p_2) = \max(1 - p_1, p_2)$.

There are two conventional notations for Boolean operators used in the definition of action effects. First, when an action has multiple effects, they will be written within a big `and` block. For example, the following definition:

```
(:action set-pqr
 :parameters (?r - robot)
 :precondition (and )
 :effect (and
   (assign (p ?r) (??f))
   (assign (q ?r) (??f))
   (assign (r ?r) (??f))
 )
)
```

Meanwhile, when a Boolean predicate is directly written in the effect, its semantics is that this Boolean state variable will be set to true. Similarly, when we write `(not (t ?r))` where `t` is a Boolean predicate, it is equivalent to `(assign (t ?r) true)`.

Boolean quantification: `forall` and `exists`. Similarly, the quantification operations are define as the following:

```
forall-expr-def: "(" "forall" (typed-variable) expr ")"
exists-expr-def: "(" "exists" (typed-variable) expr ")"
```

As an example, the following expression:

```
(exists (?o - item) (is-red ?o))
```

computes whether there is an item in the environment that is red.

Similarly, the implementation of Boolean quantifiers are: $\forall x.p(x) = \min_x p(x)$ and $\exists x.p(x) = \max_x p(x)$.

Assignment. There is only one assignment operation

```
assign-expr-def: "(" "assign" "(" predicate-name
                                  variable-list ")" expr ")"
```

The first term is the target state variable, while the second term is the expression of the value. For example, `(assign (p ?r) (??f))` means that the value of `(p ?r)` will be assigned to the return value of function `f`.

Foreach and condition. There are two special operators: for-each and condition.

```
foreach-expr-def: "(" "foreach" (typed-variable) expr ")"
when-expr-def:    "(" "when" expr expr ")"
```

When they are used in a precondition or an expression (in contrast to being under the effect definition), they means:

- `foreach`: it selects all objects of the specified type. See the previous definition of action `forward` as an example.
- `when`: it specifies the scenario in which a state variable is relevant to the computation. See the previous definition of action `forward` as an example.

When they are used in effect definitions, they means:

- `foreach`: it applies the effect formula to objects of the specified type. See the previous definition of action `mysterious-action` as an example.

- `when`: it indicates a conditional effect. The effect will be applied if and only if the condition is true. See the previous definition of action `mysterious-action` as an example.

Syntax sugars. PDSketch includes the following syntax sugars:
`(p::assign ?a1 ?a2 ... VALUE)` is equivalent to `(assign (p ?a1 ?a2 ...) VALUE)`.
`(p::cond-assign ?a1 ?a2 ... COND VALUE)` is equivalent to `(when COND (assign (p ?a1 ?a2 ...) VALUE))`.
`(p::cond-select ?a1 ?a2 ... COND)` is equivalent to `(when COND (p ?a1 ?a2 ...))`.

Blank notation ??. The **??** operation can be used to replace any expressions. It takes the following syntax:

```
slot-expr-def: "(" "??" predicate-name kwargs expr* ")"
```

For example, the definition: `(??f (item-feature ?o))` defines an function that will be externally implemented. It has name `f`. It takes only one input, which is `(item-feature ?o)`.

K.3 Domain-Independent Heuristic

In this paper, we focus on the hFF heuristic. Based on relaxed operators, it heuristically selects a set of operators that should be applied to accomplish the goal. Shown in Fig. K-1, hFF sequentially tries to apply operators that yield novel values of the features. Once the goal condition is satisfied, it back-traces the used operators. Note that such computation is general and applies to domains with arbitrarily complex feature dependencies and number of objects. We present details of our implementation in the supplementary. However, such heuristics does not naturally work with vector embeddings and neural network predictors. Next, we talk about how to "compile" trained neural features and modules into hFF-compatible representations.

In this paper, we explore two strategies for performing this reduction. In the first, we simply change all the action models so that the result is to set all of the predicate instances that they effect to have a special value called *optimistic*, which is assumed to satisfy any goal

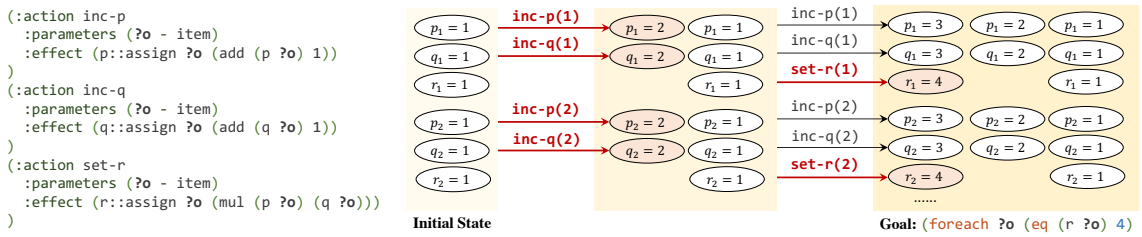


Figure K-1: **A graphical illustration of the hFF heuristics.** The goal is to set the `r` value of all objects to 4. The heuristic value hFF = 6, which is the number of selected operators (highlighted in red).

test on that predicate instance. (So, for example, after we have placed an object somewhere, we immediately "believe" it could be anywhere.) An alternative strategy, which is more computationally complex but yields "tighter" (and therefore more helpful) heuristic values, is to explicitly discretize the value space of the predicates and directly reduce to the standard discrete case. Both of these strategies are described in more detail in the supplementary material. It is important to note that these methods are inherently lossy on non-Boolean predicate values, but since they are only used for search guidance, and not in the forward simulation of the actual actions during planning, they do not affect the correctness of the overall algorithm.

Optimistic compilation: leveraging locality. The first approach, optimistic compilation (OPT), compiles each non-Boolean state variable, e.g., `(color ?o)`, into a boolean predicate `color-opt`. Any operator that changes the value of `color` (e.g., `press-button`) will set `color-opt` to true. Meanwhile, any Boolean expressions, such as `(?? (color item#1))` will return true as long as `(color-opt item#1)` is true. Intuitively, any operator that changes feature values will make the change *optimistically*.

The optimistic compilation ignores the fine-grained structure inside each state variables, but does leverage the object-level locality. For example, in the case of `(forall ?o (eq (r ?o) 4)`, it will select two operators `set-r(1)` and `set-r(2)`, leading to hFF = 2.

And-Or compilation. The And-Or compilation (AO) works in two steps. First, it discretizes all continuous state variables (poses, etc.) into a designated number of bins (e.g., 128). Next, for each externally-defined functions, it learns a first-order decision tree to approximate the computation (e.g., to approximate a neural network). Concretely, we use VQVAE [Van Den Oord *et al.*, 2017] to discretize the feature vectors. For all predicates whose output is a latent embedding, e.g., the `(item-feature ?o)` we introduced in our earlier example, we add a vector quantization layer after the encoding layers. We initialize the quantized embeddings by running a K-means clustering over the item feature embedding

from a small dataset, and finetune the weights on the entire training dataset for one epoch.

For all predicates whose inputs and outputs are both continuous parameters. We use the FOIL algorithm, the first-order decision tree learning algorithm [Quinlan, 1990] to extract first-order decision rules. We need to use FOIL instead of a plain (propositional) decision tree learning algorithm because the input to decision functions may have a variable number of objects. Here we are showing some examples of the quantized predicates:

```
(is-red ?o - item) = (or
  (item-feature@3  ?o)
  (item-feature@14 ?o)
  (item-feature@16 ?o)
  (item-feature@10 ?o)
)
```

Below we are showing a slightly more complex example, corresponding to the learned rule for action "forward."

```
(((robot-pose ?r - robot) <- (SAS
  37 <- (or
  (and
    (not (robot-direction@2 ?r - robot))
    (not (robot-direction@3 ?r - robot))
    (not (robot-direction@1 ?r - robot))
    (robot-pose@37 ?r - robot)
  )
  (and (robot-direction@3 ?r - robot) (robot-pose@45 ?r - robot))
  (and (robot-pose@36 ?r - robot) (robot-direction@0 ?r - robot))
  (and (robot-pose@29 ?r - robot) (robot-direction@1 ?r - robot))
)
  4 <- (and (robot-pose@12 ?r - robot) (robot-direction@3 ?r - robot))
  40 <- (and (robot-pose@41 ?r - robot) (robot-direction@2 ?r - robot))
  21 <- (and (robot-pose@29 ?r - robot) (robot-direction@3 ?r - robot))
  32 <- (and (robot-pose@33 ?r - robot) (robot-direction@2 ?r - robot))
  3 <- (and
```

```
    (robot-pose@11 ?r - robot) (robot-direction@3 ?r - robot))
    (and
      (not (robot-pose@27 ?r - robot))
      (not (exists (_t0 - item) (and (robot-is-facing ?r - robot _t0 - item)
                                     (item-feature@9 _t0 - item))))
      (not (exists (_t0 - item) (and (robot-is-facing ?r - robot _t0 - item)
                                     (item-feature@0 _t0 - item))))
      (not (exists (_t0 - item) (and (robot-is-facing ?r - robot _t0 - item)
                                     (item-feature@6 _t0 - item))))
      (not (exists (_t0 - item) (and (robot-is-facing ?r - robot _t0 - item)
                                     (item-feature@11 _t0 - item))))
; ... more rules omitted.
```

Note in the last section of the shown rules, the decision rule is written in first-order logic, where we need to quantify over objects.

The AO compilation further leverages the fine-grained structures of state variables. For example, the real value of features `p`, `q`, and `r` will be discretized into bins, such as `p=1`, `p=2`, `q=1`, `q=2`. A decision tree will be used to represent the transition model. Thus, the resulting model will support fine-grained simulation and back-tracing.

The role of sparsity and locality structures in heuristics. Although the discretization and heuristic computation itself does not assume any particular structure of the action definitions, their performance does rely on the programmed sparsity and locality structures, in particular, the "factorization" structure of the feature representation.

Specifically, without factorization, the entire state is described with one single vector embedding. This introduces a exponential number of possible states of the state. For example, let's assume an object state can be factorized into its x and y position (7 possible states per dimension), the color (6 possible states), and the shape (4 possible shapes). With factorization, we only need 24 possible state codes to represent each object. However, without factorization, we need $7 \times 7 \times 6 \times 4 = 1176$ states. Recall that, during the computation of heuristics, we will be solving a relaxed version of the planning program. If there is no factorization, the search problem reduces to a plain path-finding problem in the graph induced by the connectivity between these states, which is very inefficient to solve.

K.4 Experimental Setup

In this section, we detail the environment setup and domain definition of two environments: BabyAI and Painting Factory.

K.4.1 BabyAI

Setup. Following the original BabyAI setup, we use a 7 by 7 grid as the physical world. The agent is initially positioned at the center, all other object locations are randomly selected.

Offline data. The data contains both successful demonstrations and unsuccessful demonstrations. For successful ones, we use the grid-world A^* search to generate the optimal trajectory from the original agent position to the target. For unsuccessful demonstrations, we first choose an incorrect object, approach the selected object, and then runs 5 steps of random walk.

Baselines. For all baselines and our models, we use the following ways to encode object states, robot positions, and facing directions.

1. object state: following the FiLM model presented in Chevalier-Boisvert *et al.* [2019], we use an integer embedding for object states.

2. object position and robot position: we directly use the raw value as the input to the neural networks. We have tried to use embedding based methods to encode the values, but have seen consistently worse performance.

3. robot facing direction: there are four directions. We use a 4 learnable embeddings for them.

Thus, the input to the baselines are: 1) the robot state embedding, 2) the task goal specification, encoded as the concatenation of three word embeddings: the verb, the adjective, and the noun, and 3) the object state embeddings.

For both baselines BC and DT, we use a two-layer graph neural network to encode the world state into a 128-dimensional vector embedding. For BC, we use a single linear layer to predict the action to take. For DT, we use a transformer layer to aggregate the history (especially encoding the cost-to-go), and use another linear layer to predict the action in the next take.

Models. For models written in PDSketch, we have three variants.

1. In PDS-Base, the robot state encodings are treated as a single vector, named `robot-feature`, and the item state encodings are also treated as a single vector (which is the concatenation of their positions and visual features). The action definition encodes no structures. For example,

```
(:action forward
 :parameters (?r - robot)
 :precondition (and )
 :effect (and
   (robot-feature::assign ?r (??f (robot-feature ?r) (item-feature ??)))
 )
)
```

2. In PDS-Abs, we have the abstraction of the idea "robot-facing." This predicate is used to define actions. For example,

```
(:action forward
 :parameters (?r - robot)
 :precondition (and )
 :effect (robot-pose::cond-assign ?r
 (??f
   (robot-pose ?r) (robot-direction-feature ?r)
   (foreach (?o - item)
     (item-feature::cond-select ?o (robot-is-facing ?r ?o))
   )
 )
 (??g (robot-pose ?r) (robot-direction-feature ?r))
 )
)
```

3. In PDS-Rob, the robot's movements have been encoded into the definition. For example,

```
; The first function returns the 2D position that the robot is facing
;   when the robot is at ?p and facing ?d.
;   This function has been implemented externally.
```

```
; The second function returns whether an object is an obstacle (so
;   it will block the robot's movement.
;   This information has also been given to the agent as a input.
(:predicates
  (facing [return_type=pose] ?p - pose ?d - direction)
  (is-obstacle ?o - item)
)
(:derived (_is-facing ?p - pose ?d - direction ?t - pose)
  (equal (facing ?p ?d) ?t)
)
(:derived (_robot-facing [return_type=pose] ?r - robot)
  (facing (robot-pose ?r) (robot-direction ?r))
)
(:derived (_robot-is-facing ?r - robot ?o - item)
  (_is-facing (robot-pose ?r) (robot-direction ?r) (item-pose ?o))
)
(:derived (_robot-facing-clear ?r - robot)
  (not (exists (?o - item) (and
    (_robot-is-facing ?r ?o)
    (is-obstacle ?o)
  ))
)
(:action forward
 :parameters (?r - robot)
 :precondition (_robot-facing-clear ?r)
 :effect (and (robot-pose::assign ?r (_robot-facing ?r)) )
)
```

K.4.2 Painting Factory

State space. The state space in the painting factory is composed of a list of objects, and a list of containers. Both objects and containers are represented as a tuple of a 3D xyz location, and a image crop. The image crop is generated by first computing the 2D bounding box of the object in the camera plain, cropping out the image patch, and resizing it to 32 by 32. The image encoder is a 3-layer convolutional neural network followed by a linear transformation layer into a 128-dimensional embedding. The xyz location are represented as three numbers,

normalized into the range [0, 1].

Action space. The action space in the painting factory contain two actions. We first show the PDSketch definition of two actions:

```
(:action move-into
 :parameters (?o - item ?c - container)
 :precondition (and )
 :effect (and
   (item-pose::assign ?o (container-pose ?c))
   (item-feature::cond-assign ?o
     (??g (container-feature ?c))  ; when the container is a bowl,
     (??h (container-feature ?c))  ; paint ?o to be the same color as ?c.
   )
 )
)
(:action move-to
 :parameters (?o - item ?p - pose)
 :precondition (and )
 :effect (and (item-pose::assign ?o ?p))
)
```

The first action, `move-into` is an action defined for each pair of item and container. It changes the pose of the item (now the item will be inside the container). When the item is placed into a bowl, it will be painted into the same color as the bowl. The second action moves the object directly to a designated position.

Offline data. Recall that our task is to place two painted objects in the target location to form a designated relationship. To generate demonstrations, we first randomly select two blocks. Next, we paint one of them and place it at the center of the target location. Then, we paint the second object to the designated color and place it with respect to the first object.

Baselines. For all baselines, we use a two-layer graph neural network (GNN) to encode the objects. We generate the action in two steps. First, for each (block, container) pair, we generate a scalar logit indicating the score of placing the block into the container. For each block, we also generate a scalar logit indicating the score of directly moving the block. We

normalize these logits with a softmax to formulate the action probability. If the chosen action is to directly move the block, we also predict the target location.

Models. The key feature in the painting factory domain is that now we need to plan for continuous actions, instead of a discrete space. This is handled using a basic task and motion planning strategy introduced in PDDLStream [Garrett *et al.*, 2020a]. We do this in two steps.

First, during training, we build a structured model for classifying geometric relationships (on, left, and right) in the following form:

$$p(a \text{ on } b) = \sigma\left(\frac{\|pose(a) - pose(b) - \theta\|_2^2 - \gamma}{\tau}\right),$$

where θ, γ, τ are learnable parameters. σ is the Sigmoid function. That is, intuitively, we say the block a is on block b if the pose of a is close to $pose(b) + \theta$. This formulation allows us to "sample" poses of block a given the pose of block b.

Thus, during performance time, based on the initial pose of all objects, we sample a set of new poses that are on, left to, and right to the initial poses. We then use these poses to ground concrete operators. This strategy is also called "incremental" sampling in sampling-based task and motion planning literature.

Appendix L

Appendix for Behavior from Language and Demonstration

This chapter provides additional details on the BLADE framework, the experiments, and qualitative examples. Section L.1 provides a detailed description of the method, including the behavior description generation, predicate generation, abstract verification, automatic predicate annotation, classifier implementation, and policy implementation. Section L.2 provides details on the simulation experiments, including the task design and baseline implementations. Section L.3 provides qualitative examples of our method and baselines. Section L.4 provides details of our setup of the real-robot experiment. Finally, Section L.5 includes a full list of the prompts for the baselines used in the simulation experiments.

L.1 BLADE Details

L.1.1 Behavior Description Generation with LLMs

In Listing L.1, we show the behavior descriptions automatically generated by the LLM for the CALVIN domain. We also show the detailed prompt to the LLM for generating the behavior description. We break down the system prompt into four parts: definitions of primitive actions (Listings L.2), definitions of predicates and environment context (Listings L.3), an in-context example (Listings L.4), and additional instructions (Listings L.5). In Listings L.6, we show one example of the specific user prompt that is used to generate the behavior description for *place-in-drawer*.

In our experiments, we find that the environment description is necessary for the LLMs to understand the context of the task. For the simulation experiment, we provide the environment description as a list of objects and brief explanations, as shown in Listings L.3. However, this description can be automatically generated using image tagging models such

as Recognize Anything [Zhang *et al.*, 2024a] or general-purpose VLMs such as GPT-4V.

L.1.2 Predicate Generation with LLMs

Our algorithms are agnostic to the source of predicates and can flexibly generate action descriptions based on the given predicates. In our main experiment, we assume that the predicates for each task domain are provided in natural language. Here, we show that given the task definition and the environment context, a LLM can automatically generate the relevant predicates for the domain. Then, we generate behavior descriptions based on the automatically generated predicates.

To generate high-quality predicates and behavior descriptions, we take the following steps. First, the LLM is provided with the list of objects in the scene and the language-paired demonstration sequence and is required to generate relevant predicates for the domain. In Listing L.7, we show an example of the input prompt to the LLM for the CALVIN domain. Second, the LLM is prompted with the generated predicates and is required to ground the predicates using the objects in the scene. This step helps the LLM eliminate errors in the first step, such as missing arguments. Finally, we instruct the LLM to remove semantically equivalent predicates and keep the most general ones. For example, *is-on-stove*(kettle) is removed by the LLM in our experiment because *is-on*(kettle, stove) is semantically equivalent. After the predicates are automatically generated and filtered, we proceed with the behavior description generation.

In Table L.1, Table L.2, and Table L.3, we compare the generated predicates with the predicates defined by the domain expert for the CALVIN, Boil Water, and Make Tea domains. We observe that the LLM is able to generate 28 out of 30 predicates that match closely with the expert-designed predicates. These predicates provide abstract representations for object states (e.g., *is-open*, *light-on*), relations between objects (e.g., *in-slider*, *in*), and robot-centric states (e.g., *holding*). The LLM incorrectly generates the predicate *next-to*($?x$, $?y$) to characterize the effects of the *push-left* and *push-right* actions, possibly due to ambiguities in the definition of the actions. For the two real-world domains, we perform additional experiments to confirm that the generated predicates and behavior descriptions can be used to generate correct task plans on all nine real-world tasks presented in Section 11.4.6.

L.1.3 Temporal Segmentation

Before the generation of behavior description, we segment each demonstration into a sequence of *contact-based primitives*. We consider seven primitives describing the interactions between the robot and other objects: *open*/*close* grippers without holding objects, *move-to*(x) which moves the gripper to an object, *grasp*(x, y) and *place*(x, y) which grasp and place

Manually Defined	Automatically Generated
rotated-left(?*x*)	*rotated-left*(?*x*)
rotated-right(?*x*)	*rotated-right*(?*x*)
lifted(?*x*)	*holding*(?*x*)
is-open(?*x*)	*is-open*(?*x*)
is-close(?*x*)	*is-closed*(?*x*)
is-turned-on(?*x*)	*light-on*(?*x*)
is-turned-off(?*x*)	*light-off*(?*x*)
is-slider-left(?*x*)	*slider-left*(?*x*)
is-slider-right(?*x*)	*slider-right*(?*x*)
is-on(?*x*, ?*y*)	*on-table*(?*x*)
is-in(?*x*, ?*y*)	*in-slider*(?*x*), *in-drawer*(?*x*)
stacked(?*x*, ?*y*)	*on*(?*x*, ?*y*)
unstacked(?*x*, ?*y*)	*clear*(?*x*)
pushed-left(?*x*)	-
pushed-right(?*x*)	-
-	*next-to*(?*x*, ?*y*)

Table L.1: **Comparison of manually defined and automatically generated predicates for CALVIN.**

Manually Defined	Automatically Generated	Automatically Generated (Grounded)
is-placed-on(?*x*, ?*y*)	*on-table*(?*x*)	*on-table*(kettle), *on-table*(pot)
is-placed-in(?*x*, ?*y*)	*in-sink*(?*x*)	*in-sink*(kettle), *in-sink*(pot)
is-blocked(?*x*)	*on-stove*(?*x*)	*on-stove*(pot)
is-turned-away(?*x*)	*faucet-over-sink*(?*x*, ?*y*)	*faucet-over-sink*(faucet, sink)
is-aligned(?*x*, ?*y*)	*faucet-over-sink*(?*x*, ?*y*)	*faucet-over-sink*(faucet, sink)
is-turned-on(?*x*)	*faucet-on*(?*x*)	*faucet-on*(faucet)
is-turned-off(?*x*)	*faucet-off*(?*x*)	*faucet-off*(faucet)
is-filled(?*x*)	*filled*(?*x*)	*filled*(kettle), *filled*(pot)
holding(?*x*)	*holding*(?*x*)	*holding*(pot)

Table L.2: **Comparison of manually defined and automatically generated predicates for the boil-water domain.**

Manually Defined	Automatically Generated	Automatically Generated (Grounded)
is-placed-on(?*x*, ?*y*)	*is-on*(?*x*, ?*y*)	*is-on*(kettle, stove)
is-cabinet-door-open(?*x*)	*is-open*(?*x*)	*is-open*(left-door), *is-open*(right-door)
is-placed-inside(?*x*, ?*y*)	*is-in-drawer*(?*x*)	*is-in-drawer*(teabag)
	is-in-kettle(?*x*)	*is-in-kettle*(teabag)
	is-in-cabinet-left(?*x*)	*is-in-cabinet-left*(kettle)
	is-in-cabinet-right(?*x*)	*is-in-cabinet-right*(kettle)
is-drawer-open(?*x*)	*is-open*(?*x*)	*is-open*(drawer)
is-left-cabinet-door-blocked(?*x*)	*is-blocking*(?*x*, ?*y*)	*is-blocking*(pot, left-door)
is-right-cabinet-door-blocked(?*x*)	*is-blocking*(?*x*, ?*y*)	*is-blocking*(pot, right-door)
-	*is-closed*(?*x*)	*is-closed*(left-door), *is-closed*(right-door)
		is-closed(drawer)
-	*is-moved-away*(?*x*)	*is-moved-away*(pot)

Table L.3: **Comparison of manually defined and automatically generated predicates for the make-tea domain.**

object x from/onto another object y, *move*(x) which moves the currently holding object x and *push*(x).

We use a set of heuristics to automatically segment the continuous trajectories using proprioception, i.e., gripper open state, and object segmentation. Specifically, *open* and *close* are directly detected by checking whether the gripper width is at the maximum or minimum value. *grasp*(x, y) and *place*(x, y) correspond to the other closing and opening gripper actions. *move*(x), *push*(x) and *move-to*(x) are matched to temporal segments between pairs of gripper actions. Their type can be inferred based on the preceding and following gripper actions. We make a simplifying assumption that the robot moves freely in space only when the gripper is fully open and pushes objects only when the gripper is fully closed. These are given as instructions to the human demonstrators. In the simulator, the arguments of the primitives are obtained from the contact state. In the real world, they are inferred from the language annotations of the actions (e.g.,"place the kettle on the stove" corresponds to *place*(kettle, stove)) procedurally or by the LLMs. The arguments can also be left unspecified; these arguments mainly provide additional contextual information about the target objects.

We have discussed that we use LLMs to predict a *body* of contact primitive sequence associated with each behavior description. This additional step helps account for noises in the segmentation annotations, which are prevalent in CALVIN's language-annotated demonstrations. For example, the language annotation "lift-block-table" correspond to the contact sequence {*move-to*, *grasp*, *move*, *place*}. Based on the generated *body*, the behavior can be correctly mapped to {*grasp*, *move*} and the demonstration trajectories can then be re-segmented. This additional step is crucial for learning accurate groundings of the states and actions.

Our approach to temporal segmentation are similar to keyframe-based methods like PerAct [Shridhar *et al.*, 2022b] and 3D Diffuser Actor [Ke *et al.*, 2024], which rely on end-effector states (e.g., grasp and release) and velocities to segment demonstrations. However,

Algorithm 14 Abstract Verification

Input: Dataset $\mathcal{D}$, Behavior descriptions $\mathcal{A}$

error_counter ← a counter for sequencing errors related to each behavior
counter ← a counter for storing the occurrences of each behavior
for $i \leftarrow 1$ to K **do**
  obtain a behavior sequence $\mathcal{D}_i \leftarrow \{a_1^i, ..., a_N^i\}$
  initialize a dictionary for predicate state *pred* ← {}
  for $t \leftarrow 1$ to N **do**
    for each *exp* in $pre_{a_t^i}$ **do**
      $(p, v) \leftarrow$ ExtractPredicateAndBool(*exp*)
      if p not in *pred* **then**
        $pred[p] \leftarrow v$
      else
        if $pred[p] \neq v$ **then**
          increment $error_counter[a_t^i]$
    for each *exp* in $eff_{a_t^i}$ **do**
      $(p, v) \leftarrow$ ExtractPredicateAndBool(*exp*)
      $pred[p] \leftarrow v$
    increment $counter[a_t^i]$
for each a in *error_counter* **do**
  if $error_counter[a]/counter[a] > threshold$ **then**
    regenerate the behavior description for a

our method differs by distinguishing between prehensile and non-prehensile manipulation through detecting whether an object is grasped. This enables the development of behaviors such as *move-faucet-away*, where the robot pushes the faucet head without grasping it. In our preliminary studies, we also experiment with other vision-based methods including UVD [Zhang *et al.*, 2024b] and Lotus [Wan *et al.*, 2024]. A main issue for incorporating these methods is that they provide less consistent segmentations for different occurrences of the same behavior. As we discussed in Chapter 11, more advanced contact detection techniques will be an important future direction for using contact primitives as a meaningful interface between actions and language.

L.1.4 Abstract Verification

After the generation of the behavior descriptions, we verify the generated behavior descriptions by performing abstract verification on the demonstration trajectories. Given a segmented sequence of the trajectory where each segment is associated with a behavior, we verify whether the preconditions of each behavior can be satisfied by the accumulated effects of the previous behaviors. Pseudocode for this algorithm is shown in Algorithm 14.

L.1.5 Automatic Predicate Annotation

We leverage *all* behavior descriptions to automatically label an observation sequence $\{o_1, ..., o_H\}$ based on its associated segmentation. In particular, at o_0, we label all state predicates as "unknown." Next, we unroll the sequence of executed behaviors. As illustrated in Fig. 11-13c, before applying a behavior a at step o_t, we label all predicates in pre_a true and predicates in eff_a false. When a finishes at step $o_{t'}$, we label all predicates in eff_a. In addition, we will propagate the labels for state predicates to later time steps until they are explicitly altered by another behavior a. Pseudocode for this algorithm is shown in Algorithm 15.

As a result, we obtain both positive and negative examples to train the binary predicate classifiers. In particular, the negative examples of a predicate come from two sources. First, the preconditions and effects of a behavior can include negated predicates. Second, during automatic predicate annotation, we label all predicates in the effects as false.

L.1.6 Classifier Implementation

Based on the state predicate dataset generated from behavior definitions, we train a set of state classifiers $f_\theta(p) : O \rightarrow \{T, F\}$, which are implemented as standard neural networks for classification.

In the simulation experiment, the classifier model is based on a pre-trained CLIP model (`ViT-B/32`). We use the image pre-processing pipeline from the CLIP model to process the input images. We use images from the static camera in the simulation. We perform one additional step of image processing to mask out the robot arm, which we find in our preliminary experiment to help avoid overfitting. We do not use the global image embedding from the CLIP model, instead we extract the patch tokens from the output of the vision transformer. We downsize the concatenated patch tokens with a multilayer perceptron (MLP) and then concatenate with word embeddings of the predicate arguments (e.g., *red-block*, *table*). The final embedding is then passed through a predicate-specific MLP to output the logit for binary classification. The CLIP model is frozen, while all other learnable parameters are trained.

In the real-world experiment, we find that, with more limited data than simulation, the pre-trained CLIP model often overfits to spurious relations in the training images (e.g., the state of the faucet is entangled with the location of the kettle). We also experiment with a ResNet-50 model pre-trained on ImageNet and find similar behavior. To improve generalization, we choose to focus on relevant objects and regions. We achieve this by using segmented object point clouds. We use open vocabulary object detector Grounding-Dino [Liu *et al.*, 2024a] to detect objects given object names. The predicted 2D bounding boxes are projected into 3D and used to extract regions of the point cloud surrounding each

Algorithm 15 Predicate Annotation

Input: Behavior sequence $\{a_1, ..., a_N\}$, Observation sequence $\{o_1, ..., o_H\}$, Descriptions $\mathcal{A}$

propagated ← an empty list of propagated predicates
prev_effs ← a list for storing effects from previous step
timed_preds ← an empty list of predicates associated with time steps
pred_obs ← an empty list for storing predicates paired with observations
for $t \leftarrow 1$ to N **do**
  // Precondition
  timed_preds ← *timed_preds* ∪ GetTimedPredicates(pre_{a_t}, t)
  timed_preds ← *timed_preds* ∪ GetTimedPredicates($\neg eff_{a_t}, t$)
  // Propagated
  for each p in *propagated* **do**
    if not altered(p, a_t) **then**
      UpdateTime(p, t)
    else
      propagated.remove(p)
      timed_preds.add(p)
  // Previous effects
  for each p in *prev_effs* **do**
    if not altered(p, a_t) **then**
      propagated.add(p)
    else
      timed_preds.add(p)
  // Store effects for next step
  prev_effs ← GetTimedPredicates(eff_{a_t}, t)
timed_preds.update(*propagated*)
timed_preds.update(*prev_effs*)
for each p in *timed_preds* **do**
  pred_obs.update(MatchTimedPredicateWithObservation(p, $\{o_1, ..., o_H\}$))
return *pred_obs*

object. The point-cloud-based classifier is based on the shape classification model from the Point Cloud Transformer (PCT) [Guo *et al.*, 2021]. We concatenate the segmented object point clouds and include one additional channel to indicate the identity of each point. The PCT is used to encode the combined point cloud and output the final logit. The PCT model is trained from scratch.

We also experiment with replacing trained classifiers a general-purpose VLM `gpt-4o`. The VLM fails to robustly determine the boolean values of the predicates in different states. Our observation is that the VLM can more reliably detect the states of articulated objects (e.g., the states of drawers and cabinet doors) than more complex spatial relations (e.g., whether the faucet head is aligned with the kettle). Following our original experimental procedure, we combine the VLM classifier with our planner to test the generalization cases for the Boil Water domain. Due to the inaccuracy of the state classification, the overall system achieves a 0% success rate.

We imagine future VLMs will become more reliable in detecting the predicate states, and our method will be beneficial to the paradigm of using VLMs as classifiers in the following ways: 1) our automatic predicate annotation method can generate examples for VLMs to recognize new concepts through visual in-context prompting or supervised fine-tuning; 2) our method provides data to train task-specific classifiers based on 3D representations, which are complementary to general-purpose VLMs in recognizing geometric and spatial concepts; 3) our method provides a way to learn user-specific predicates (e.g., a predicate that determines whether clean dishes are arranged according to a user's preferences) from demonstrations. In our preliminary experiment on the Boil Water domain, we find that using examples generated from our method as in-context visual examples improves the VLM's accuracy by around 5% on 220 withheld testing examples.

L.1.7 Policy Implementation

For each behavior, we train control policies $\pi_\theta(a) : \mathcal{O} \rightarrow \mathcal{U}$, implemented as a diffusion policy [Chi *et al.*, 2023]. We make three changes to the original implementation to facilitate chaining the learned behaviors. First, when training the model to predict the first raw action for each skill, we replace the history observations with observations sampled randomly from a temporal window prior to when the skill is executed, to avoid bias in the starting positions of the robot arm. Second, we perform biased sampling of the training sequences to ensure that the policy is trained on a diverse set of starting positions. Third, at the end of each training sequence, we append a sequence of zeros actions so the learned policy can learned to predicate termination. These strategies are implemented for both the simulation and the real world.

In simulation, we construct the point cloud of the scene using the RGB-D image

from the frame-mounted camera. We then obtain segmented object point clouds for the relevant objects of each behavior (e.g., *table* and *block* for *pick-block-table*) with groundtruth segmentation masks from the PyBullet simulator. The segmented point clouds of the objects are concatenated to form the input point cloud observation. The model uses the PCT to encode a sequence of point clouds as history observations and uses another time-series transformer encoder to reason over the history observations and predict the next actions. The time-series transformer is similar in design to the transformer-based diffusion policy [Chi *et al.*, 2023].

In the real world, we use RGB images from four stationary cameras mounted around the workspace and a wrist-mounted camera as input to an image-based diffusion policy model. The input is processed using five separate ResNet-34 encoder heads. The policy directly predicts the gripper pose in the world frame. We found the wrist-mounted camera to be particularly helpful in the real-world setup.

L.1.8 Planner Implementation

Planning over geometric constraints. Geometric constraints, specifically the collision-free constraints for each action, are handled "in the now," right before an action is executed. This is because in order to classify the geometric constraints, we would need to know the exact pose of all objects in the environments. However, we do not explicitly learn models for predicting the exact location of objects after executing certain behaviors.

Our approach to handle this is to process them in the now. It follows the hierarchical planning strategy [Kaelbling and Lozano-Pérez, 2011]. In particular, the precondition for actions is an ordered list. In our case, there are two levels: the second level contains the geometric constraint preconditions and the first level contains the rest of the semantic preconditions. During planning, only the first set of preconditions will be added to the subgoal list. After we have finished planning for the first-level preconditions, we consider the second-level precondition for the first behavior in the resulting plan, by possibly moving other obstacles away.

As an example, let us consider the skill of opening the cabinet door. Its first-level precondition only considers the initial state of the cabinet door (i.e., it should be initially closed). It also has a second-level precondition stating that nothing else should be blocking the door. In the beginning, the planner only considers the first-level preconditions. When this behavior is selected to be executed next, the planner checks for the second-level precondition. If this non-blocking precondition is not satisfied in the current state, we will recursively call the planner to achieve it (which will generate actions that move the blocking obstacles away). If this precondition has already been satisfied, we will proceed to execute the policy associated with this *opening-cabinet-door* skill.

This strategy will work for scenarios where there is enough space for moving obstacles around and the robot does not need to make dedicated plans for arranging objects. In scenarios where space is tight and dedicated object placement planning is required, we can extend our framework to include the prediction of object poses after each skill execution.

Planning over partial observability. Partial observability is handled assuming the most likely state. In particular, the effect definitions for all behaviors are deterministic. It denotes the most likely state that it will result in. For example, in the definition of behaviors for finding objects (e.g., the *find-object-in-left-cabinet*), we have a deterministic and "optimistic" effect statement that the object will be visible after executing this action.

At performance time, since we will replan after executing each behavior, if the object is not visible after we have opened the left cabinet, the planner will automatically plan for other actions to achieve this visibility subgoal.

This strategy works for simple partially observable Markov decision processes (POMDPs). A potential extension to it is to model a belief state (e.g., representing a distribution of possible object poses) and execute belief updates on it. Planners can then use more advanced algorithms such as observation-based planning to generate plans. Such strategies have been studied in task and motion planning literature [Kaelbling and Lozano-Pérez, 2011; Garrett *et al.*, 2020b].

L.2 Simulation Experiment Details

In the effort to standardize our evaluation, we adopt the standard CALVIN benchmark to evaluate all methods without any modifications to its language annotations, action space, skills, and environments.

L.2.1 Task Design

To evaluate generalization to new long-horizon manipulation tasks, we designed six tasks that fall into three categories: Abstract Goal, Geometric Constraint, and Partial Observability. Each task has a language instruction, a sampler that generates random initial states, and a goal satisfaction function for evaluation. We provide details for each task below.

Task-1

- **Task Category:** Abstract Goal
- **Language Instruction:** *turn off all lights.*
- **Logical Goal:** (and (is-turned-off led) (is-turned-off lightbulb))

- **Initial State:** Both the led and the lightbulb are initially turned on.
- **Goal Satisfaction:** The logical states of both the lightbulb and the led are off.
- **Variation:** The initial states of the led and the lightbulb are both on and the goal is to turn them off.

Task-2

- **Task Category:** Abstract Goal
- **Language Instruction:** *move all blocks to the closed drawer.*
- **Logical Goal:** (and (is-in red-block drawer) (is-in blue-block drawer) (is-in pink-block drawer))
- **Initial State:** The blocks are visible and not in the drawer. The drawer is closed.
- **Goal Satisfaction:** The blocks are in the drawer.

Task-3

- **Task Category:** Abstract Goal
- **Language Instruction:** *move all blocks to the open drawer.*
- **Logical Goal:** (and (is-in red-block drawer) (is-in blue-block drawer) (is-in pink-block drawer))
- **Initial State:** The blocks are visible and not in the drawer. The drawer is open.
- **Goal Satisfaction:** The blocks are in the drawer.

Task-4

- **Task Category:** Partial Observability
- **Language Instruction:** *place a red block on the table.*
- **Logical Goal:** (is-on red-block table)
- **Initial State:** The red block is in the drawer and the drawer is closed.
- **Goal Satisfaction:** The red block is placed on the table.
- **Variations:** Find the blue block or the pink block.

Task-5

- **Task Category:** Partial Observability
- **Language Instruction:** *place a red block on the table.*
- **Logical Goal:** (is-on red-block table)
- **Initial State:** The red block is behind the sliding door.
- **Goal Satisfaction:** The red block is placed on the table.
- **Variations:** Find the blue block or the pink block.

Task-6

- **Task Category:** Geometric Constraint
- **Language Instruction:** *open the slider.*
- **Logical Goal:** (is-slider-left slider)
- **Initial State:** The sliding door is on the right and there is a pink block on the path of the sliding door to the left.
- **Goal Satisfaction:** The sliding door is within 5cm of the left end.
- **Variations:** Move the slider to the right.

L.2.2 Baseline Implementation

HULC. This baseline is a hierarchical policy learning method that learns from language-annotated play data using hindsight labeling [Mees *et al.*, 2022]. It's one of the best-performing models on the $D \rightarrow D$ split of the CALVIN benchmark. We omit the comparison to the HULC++ method [Mees *et al.*, 2023], the follow-up work of HULC that leverages affordance prediction and motion planning to improve the low-level skills, because our evaluation is focused on the task planning ability of the learned hierarchical model.

SayCan. This baseline combines an LLM-based planner that takes the language instruction and learned feasibility functions for skills to perform task planning. SayCan relies on well-trained affordance models to filter out inadmissible actions from the LLMs. Because we learn new skills from demonstrations, it is infeasible to learn Q-functions for these skills through RL. We opt to use two other types of affordance functions*. The first affordance function is based on the detected objects in the scene.. We provide the detected objects in the prompt. The second affordance function is an image-based neural network that predicts the likelihood of taking an action in a physical state, and the network is trained on our demonstration data. The backbone of the neural network is the same as our image-based predicate classifiers. The prompt of the model is listed in Listing L.8.

Robot-VILA. This baseline performs task planning with a VLM. We adopt the prompts provided in the original paper to the CALVIN environment. The prompts are divided into the initial prompt that is used to generate the task plan given the initial observation (shown in Listing L.9) and the follow-up prompt that is used for all subsequent steps (shown in Listing L.10). We use `gpt-4-turbo-2024-04-09` as the VLM. Because the model does not memorize the history. We store the history dialogue, including the text input and the

*We closely follow the official open-source SayCan implementation, available at https://github.com/google-research/google-research/tree/master/saycan.

image input, and concatenate the history dialogue with the current dialogue as the input to the VLM.

T2M-Shooting. This baseline (in particular, the shooting-based algorithm) is similar to the SayCan algorithm except that: 1) it uses a multi-step feasibility model in contrast to the single-step feasibility model used by SayCan; 2) the LLM additionally takes a symbolic state description of object states and relationships. The original Text2Motion method assumes access to ground-truth symbolic states. For comparison, in this paper, we compare Text2Motion with BLADE in two settings: one with the ground-truth states and the other with the state classifiers learned by BLADE. The prompt of the model is listed in Listing L.11.

L.3 Qualitative Examples

In this section, we include three qualitative examples from the CALVIN experiments to compare the generalization capabilities of BLADE with baselines. Specifically, Fig. L-4 shows generalization to abstract goal, Fig. L-5 shows generalization to partial observability, and Fig. L-6 shows generalization to geometric constraint. In summary, BLADE is able to generate accurate long-horizon manipulation plans for novel situations while latent planning, LLM, and VLM baselines fail.

L.4 Real World Experiment Details

We validated our approach in two real-world domains. As shown in Fig. L-1, we employ a 7-degree of freedom (DOF) Franka Emika robotic arm equipped with a parallel jaw gripper. A total of Five Intel RealSense RGB-D cameras are used to provide observation for our policies and state classifiers. Four cameras are mounted on the frame and one additional camera is mounted on the robot's wrist.

Our teleoperation system uses a 3DConnexion SpaceMouse for control. During the collection of demonstrations, we record the the pose of the end effector, the gripper width, and the RGB-D images from the five cameras. We collected 85 demonstrations for each of the two real-world domains, which provide the training data for the diffusion policy models and the state classifiers.

L.4.1 Task Design

Similar to our simulation experiments, our evaluation protocol includes the design of six tasks aimed at assessing the model's generalization capabilities across new long-horizon

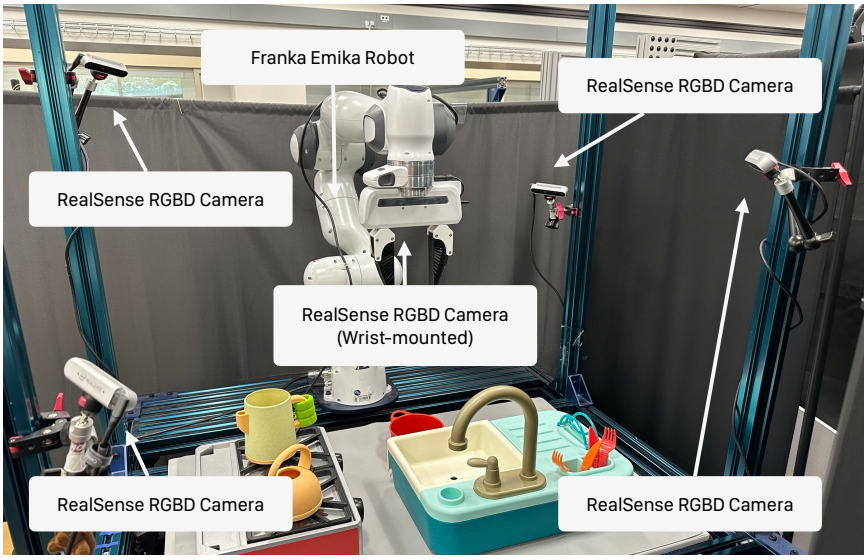


Figure L-1: **Our real-robot setup.** We use a 7-degree of freedom (DOF) Franka Emika robotic arm with a parallel jaw gripper for our real-world experiment. A total of Five Intel RealSense RGB-D cameras are used to provide observation for our policies and state classifiers. Four cameras are mounted on the frame and an additional one is mounted to the robot's wrist.

tasks. These tasks are specifically crafted to test the model's proficiency for four types of generalization: Unseen Initial Condition, State Perturbation, Partial Observability, and Geometric Constraint.

Task-1

- **Domain:** Boil Water
- **Task Category:** Unseen Initial Condition
- **Language Instruction:** *Fill the kettle with water and place it on the stove*
- **Logical Goal:** (and (is-filled kettle) (is-placed-on kettle stove) (is-turned-off faucet-knob))
- **Initial State:** The kettle is placed inside the sink, and the stove is not blocked. The faucet is turned off with the faucet head turned away.

Task-2

- **Domain:** Boil Water
- **Task Category:** State Perturbation
- **Language Instruction:** *Fill the kettle with water and place it on the stove*
- **Logical Goal:** (and (is-filled kettle) (is-placed-on kettle stove) (is-turned-off faucet-knob))

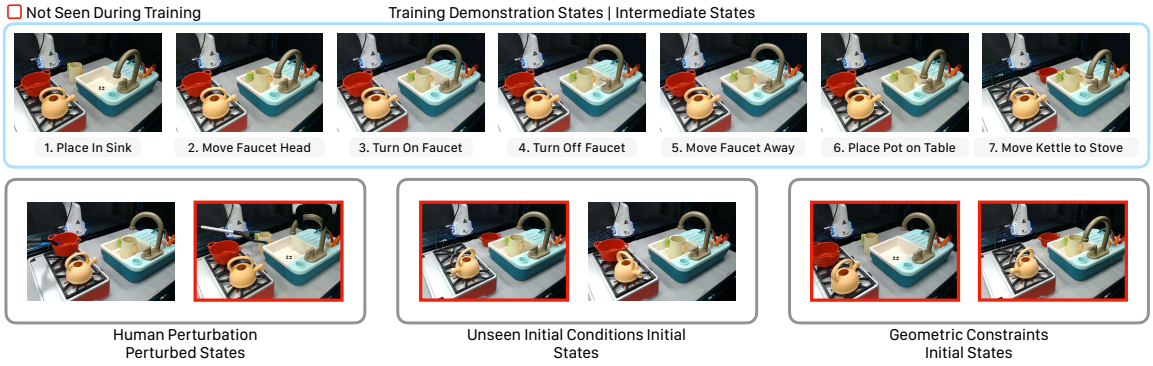


Figure L-2: **Training and testing states for the boil-water domain.**

- **Initial State:** The kettle is placed inside the sink and the stove is blocked.
- **Perturbation**: The human user moves the kettle from the sink to the table after the robot turns the faucet head towards the sink. The robot needs to replan to move the kettle back to the sink.

Task-3

- **Domain:** Boil Water
- **Task Category:** Geometric Constraint
- **Language Instruction:** *Fill the kettle with water and place it on the stove*
- **Logical Goal:** (and (is-filled kettle) (is-placed-on kettle stove) (is-turned-off faucet-knob))
- **Initial State:** The kettle is placed inside the sink and the stove is blocked, creating a geometric constraint.

Task-4

- **Domain:** Make Tea
- **Task Category:** Unseen Initial Condition
- **Language Instruction:** *Place the kettle on the stove and place the teabag inside the kettle.*
- **Logical Goal:** (and (is-placed-on kettle stove) (is-placed-inside teabag kettle))
- **Initial State:** The kettle is placed inside a cabinet. The cabinet doors are open. The drawer is closed.

Task-5

- **Domain:** Make Tea
- **Task Category:** State Perturbation

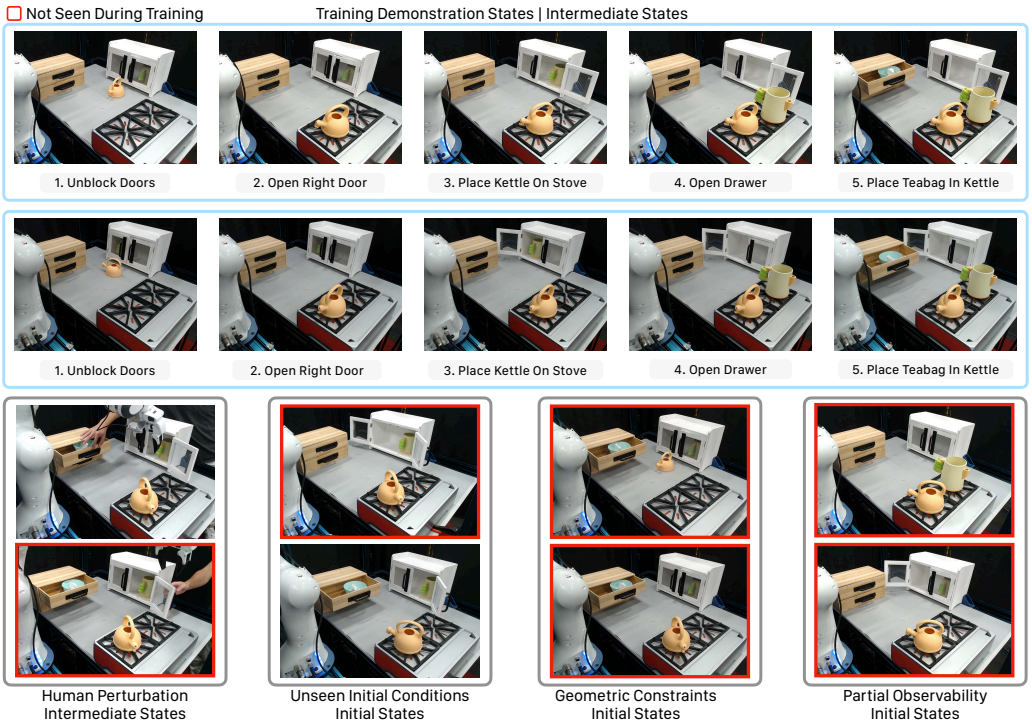


Figure L-3: **Training and testing states for the make-tea domain.**

- **Language Instruction:** *Place the kettle on the stove and place the teabag inside the kettle.*
- **Logical Goal:** (and (is-placed-on kettle stove) (is-placed-inside teabag kettle))
- **Initial State:** The kettle is placed inside the cabinet and the cabinet door is open. The drawer is initially closed.
- **Perturbation**: Once the robot opens the drawer, a human user closes the drawer.

Task-6

- **Domain:** Make Tea
- **Task Category:** Geometric Constraint
- **Language Instruction:** *Place the kettle on the stove and place the teabag inside the kettle.*
- **Logical Goal:** (and (is-placed-on kettle stove) (is-placed-inside teabag kettle))
- **Initial State:** There is a teapot blocking the cabinet doors. The kettle is inside the cabinet. The drawer is open with the teabag visible.

Task-7

- **Domain:** Make Tea
- **Task Category:** Partial Observability

- **Language Instruction:** *Place the kettle on the stove and place the teabag inside the kettle.*
- **Logical Goal:** (and (is-placed-on kettle stove) (is-placed-inside teabag kettle))
- **Initial State:** The kettle is placed inside a cabinet and is not visible.

L.4.2 Qualitative Examples of Novel States

In Fig. L-2 and Fig. L-3, we visualize and confirm that more than half of the initial states and the perturbed states are not part of the demonstrations in our experiments; therefore, purely imitation-learning-based methods will struggle to solve.

L.5 Prompts for Baselines

In this section, we provide the prompts for the baselines used in the simulation experiments. We provide the prompts for SayCan in Listing L.8, Robot-VILA in Listing L.9 and Listing L.10, and T2M-Shooting in Listing L.11.



Figure L-4: **BLADE and baseline performance on an abstract goal generalization task in the CALVIN environment.**

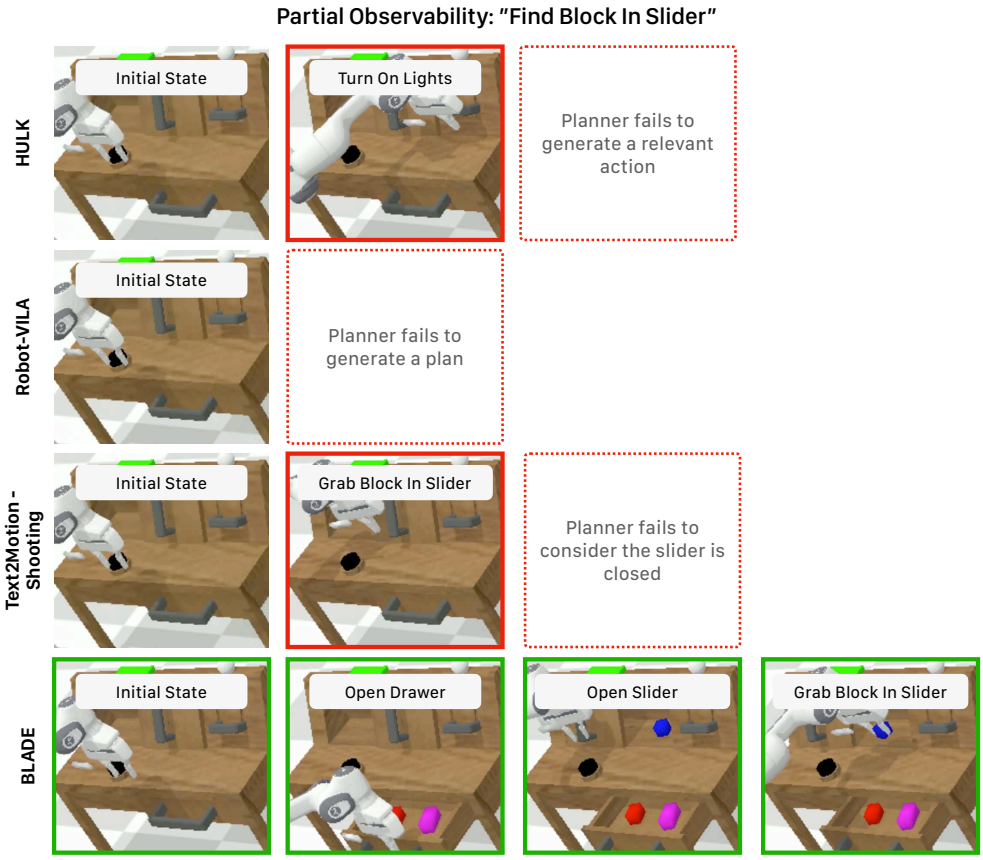


Figure L-5: **BLADE and baseline performance on the partial observability generalization task in the CALVIN environment.**

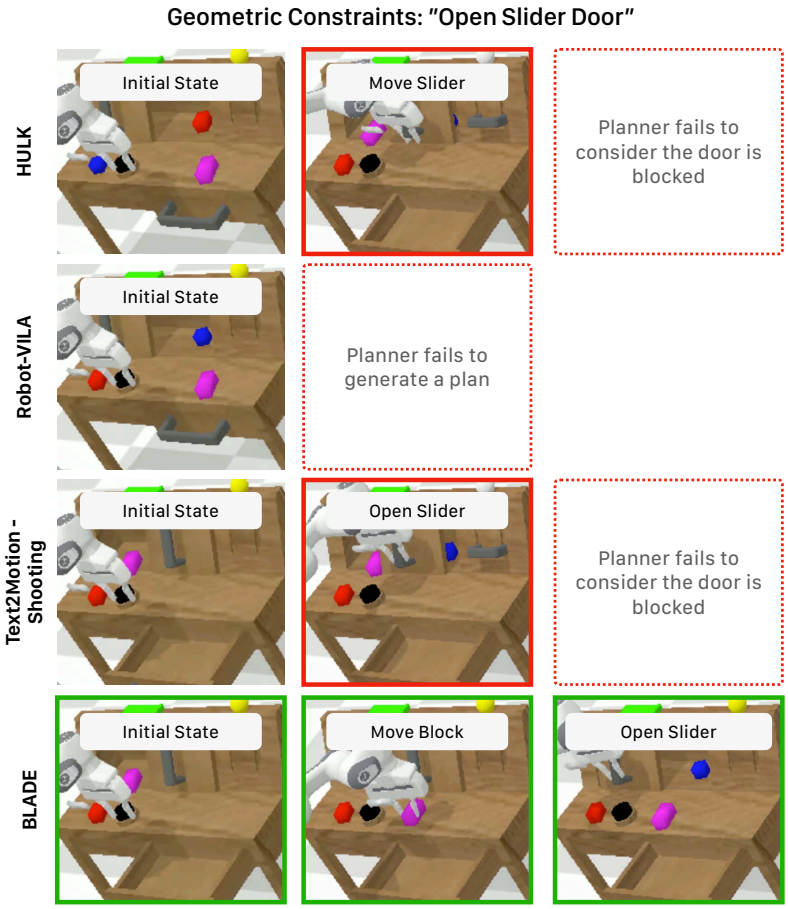


Figure L-6: **BLADE and baseline performance on the geometric constraint generalization task in the CALVIN environment.**

Listing L.1: **Behavior descriptions generated by the LLM for the CALVIN domain.**

```
;; lift_block_table
(:action lift-block-table
 :parameters (?block - item ?table - item)
 :precondition (and (is-block ?block) (is-table ?table) (is-on ?block ?table) (not (is-
 lifted ?block)))
 :effect (and (lifted ?block) (not (is-on ?block ?table)))
 :body (then
   (grasp ?block ?table)
   (move ?block)
 )
)

;; lift_block_slider
(:action lift_block_slider
 :parameters (?block - item ?slider - item)
 :precondition (and (is-block ?block) (is-slider ?slider) (is-in ?block ?slider))
 :effect (and (lifted ?block) (not (is-in ?block ?slider)))
 :body (then
   (grasp ?block ?slider)
   (move ?block)
 )
)

;; lift_block_drawer
(:action lift-block-drawer
 :parameters (?block - item ?drawer - item)
 :precondition (and (is-block ?block) (is-drawer ?drawer) (is-in ?block ?drawer) (is-open
 ?drawer))
 :effect (and (lifted ?block) (not (is-in ?block ?drawer)))
 :body (then
   (grasp ?block ?drawer)
   (move ?block)
 )
)

;; place_in_slider
(:action place-in-slider
 :parameters (?block - item ?slider - item)
 :precondition (and (is-block ?block) (is-slider ?slider) (is-lifted ?block))
 :effect (and (is-in ?block ?slider) (not (is-lifted ?block)))
 :body (then
   (place ?block ?slider)
 )
)

;; place_in_drawer
(:action place-in-drawer
 :parameters (?block - item ?drawer - item)
 :precondition (and (is-block ?block) (is-drawer ?drawer) (is-lifted ?block) (is-open ?
 drawer))
 :effect (and (is-in ?block ?drawer) (not (is-lifted ?block)))
 :body (then
   (place ?block ?drawer)
 )
)
```

```
;; place_on_table
(:action place-on-table
 :parameters (?block - item ?table - item)
 :precondition (and (is-block ?block) (is-table ?table) (is-lifted ?block))
 :effect (and (is-on ?block ?table) (not (is-lifted ?block)))
 :body (then
   (place ?block ?table)
 )
)

;; stack_block
(:action stack_block
 :parameters (?block - item ?target - item)
 :precondition (and (is-block ?block) (is-block ?target) (is-lifted ?block))
 :effect (and (stacked ?block ?target) (not (is-lifted ?block)))
 :body (then
   (place ?block ?target)
 )
)


;; unstack_block
(:action unstack_block
 :parameters (?block1 - item ?block2 - item)
 :precondition (and (is-block ?block1) (is-block ?block2) (stacked ?block1 ?block2))
 :effect (and (unstacked ?block1 ?block2) (is-lifted ?block1) (not (stacked ?block1 ?
 block2)))
 :body (then
   (grasp ?block1 ?block2)
   (move ?block1)
 )
)

;; rotate_block_right
(:action rotate-block-right
 :parameters (?block - item ?table - item)
 :precondition (and (is-block ?block) (is-table ?table) (is-on ?block ?table))
 :effect (and
           (rotated-right ?block)
           (not (rotated-left ?block)))
 :body (then
   (grasp ?block ?table)
   (move ?block)
   (place ?block ?table)
 )
)

;; rotate_block_left
(:action rotate_block_left
 :parameters (?block - item ?table - item)
 :precondition (and (is-block ?block) (is-table ?table) (is-on ?block ?table))
 :effect (and (rotated-left ?block))
 :body (then
   (grasp ?block)
   (move ?block)
   (place ?block)
```

```
 )
)

;; push_block_right
(:action push_block_right
 :parameters (?block - item ?table - item)
 :precondition (and (is-block ?block) (is-table ?table) (is-on ?block ?table))
 :effect (and (pushed-right ?block) (not (pushed-left ?block)))
 :body (then
   (close)
   (push ?block)
   (open)
 )
)

;; push_block_left
(:action push-block-left
 :parameters (?block - item)
 :precondition (and (is-block ?block))
 :effect (and (pushed-left ?block))
 :body (then
   (close)
   (push ?block)
   (open)
 )
)

;; move_slider_left
(:action move_slider_left
 :parameters (?slider - item)
 :precondition (and (is-slider ?slider) (is-slider-right ?slider))
 :effect (and (is-slider-left ?slider) (not (is-slider-right ?slider)))
 :body (then
   (grasp ?slider)
   (move ?slider)
   (place ?slider)
 )
)

;; move_slider_right
(:action move-slider-right
 :parameters (?slider - item)
 :precondition (and (is-slider ?slider) (not (is-slider-right ?slider)))
 :effect (and (is-slider-right ?slider))
 :body (then
   (grasp ?slider)
   (move ?slider)
   (place ?slider)
 )
)

;; open_drawer
(:action open-drawer
 :parameters (?drawer - item)
 :precondition (and (is-drawer ?drawer) (is-close ?drawer))
 :effect (and (is-open ?drawer) (not (is-close ?drawer)))
 :body (then
```

```
   (close)
   (push ?drawer)
   (open)
 )
)

;; close_drawer
(:action close-drawer
 :parameters (?drawer - item)
 :precondition (and (is-drawer ?drawer) (is-open ?drawer))
 :effect (and (is-close ?drawer) (not (is-open ?drawer)))
 :body (then
   (close)
   (push ?drawer)
   (open)
 )
)

;; turn_on_lightbulb
(:action turn-on-lightbulb
 :parameters (?lightbulb - item)
 :precondition (and (is-lightbulb ?lightbulb) (is-turned-off ?lightbulb))
 :effect (and (is-turned-on ?lightbulb) (not (is-turned-off ?lightbulb)))
 :body (then
   (close)
   (push ?lightbulb)
   (open)
 )
)

;; turn_off_lightbulb
(:action turn-off-lightbulb
 :parameters (?lightbulb - item)
 :precondition (and (is-lightbulb ?lightbulb) (is-turned-on ?lightbulb))
 :effect (and (is-turned-off ?lightbulb) (not (is-turned-on ?lightbulb)))
 :body (then
   (close) (push ?lightbulb) (open)
 )
)

;; turn_on_led
(:action turn-on-led
 :parameters (?led - item)
 :precondition (is-led ?led)
 :effect (and (is-turned-on ?led) (not (is-turned-off ?led)))
 :body (then
   (close)
   (push ?led)
   (open)
 )
)

;; turn_off_led
(:action turn-off-led
 :parameters (?led - item)
 :precondition (and (is-led ?led) (is-turned-on ?led))
 :effect (and (is-turned-off ?led) (not (is-turned-on ?led)))
```

```
 :body (then
   (close)
   (push ?led)
   (open)
 )
)

;; push_into_drawer
(:action push-into-drawer
 :parameters (?block - item ?drawer - item)
 :precondition (and (is-block ?block) (is-drawer ?drawer) (is-open ?drawer))
 :effect (and (is-in ?block ?drawer))
 :body (then
   (close)
   (push ?block)
   (open)
 )
)
```

Listing L.2: **Example Prompt for CALVIN–Contact Primitives.**

```
**Primitive Actions:**
There are seven primitive actions that the robot can perform. They are:
- (grasp ?x ?y): ?x and ?y are two object variables. ?x is the object that the robot will
be grasping, ?y is the object that ?x is currently on or in.
- (place ?x ?y): ?x and ?y are two object variables. ?x is the object that the robot is
currently holding, ?y is the object that ?x will be placed on or in.
- (move ?x): ?x is the object that the robot is currently holding and will be moved by the
 robot.
- (push ?x): ?x is the object that the robot will be pushing.
- (move-to ?x): the robot arm will move without holding any object or pushing any object.
- (open): the robot gripper will open fully.
- (close): the robot gripper will close without grasping any object.

**Combined Primitives:**
The primitive actions can be combined into a high-level routine. For example, (then (grasp
 ?x ?y) (move ?x) (place ?x ?y)) means the robot will pick up ?x from ?y, move ?x, and
place ?x to ?z. The possible combination of primitives are:
A. (then (grasp ?x ?y) (move ?x))
B. (then (place ?x ?y))
C. (then (grasp ?x ?y) (move ?x) (place ?x ?z))
D. (then (close) (push ?x) (open))
```

Listing L.3: **Example Prompt for CALVIN–Environment.**

```
**Predicates for Preconditions and Effects:**
The list of all possible predicates for defining the preconditions and effects of the high
-level routine are listed below:

For specifying the type of the object:
- (is-table ?x - item): ?x is a table.
- (is-slider ?x - item): ?x is a slider.
- (is-drawer ?x - item): ?x is a drawer.
- (is-lightbulb ?x - item): ?x is a lightbulb.
- (is-led ?x - item): ?x is a led.
- (is-block ?x - item): ?x is a block.
```

```
For specifying the attributes of the object:
- (is-red ?x - item): ?x is red. This predicate applies to a block.
- (is-blue ?x - item): ?x is blue. This predicate applies to a block.
- (is-pink ?x - item): ?x is pink. This predicate applies to a block.

For specifying the state of the object:
- (rotated-left ?x - item): ?x is rotated left. This predicate applies to a block.
- (rotated-right ?x - item): ?x is rotated right. This predicate applies to a block.
- (pushed-left ?x - item): ?x is pushed left. This predicate applies to a block.
- (pushed-right ?x - item): ?x is pushed right. This predicate applies to a block.
- (lifted ?x - item): ?x is lifted in the air. This predicate applies to a block.
- (is-open ?x - item): ?x is open. This predicate applies to a drawer.
- (is-close ?x - item): ?x is close. This predicate applies to a drawer.
- (is-turned-on ?x - item): ?x is turned on. This predicate applies to a lightbulb or a
led.
- (is-turned-off ?x - item): ?x is turned off. This predicate applies to a lightbulb or a
led.
- (is-slider-left ?x - item): the sliding door of the slider ?x is on the left.
- (is-slider-right ?x - item): the sliding door of the slider ?x is on the right.

For specifying the relationship between objects:
- (is-on ?x - item ?y - item): ?x is on top of ?y. This predicate applies when ?x is a
block and ?y is a table.
- (is-in ?x - item ?y - item): ?x is inside of ?y. This predicate applies when ?x is a
block and ?y is a drawer or a slider.
- (stacked ?x - item ?y - item): ?x is stacked on top of ?y. This predicate applies when ?
x and ?y are blocks.
- (unstacked ?x - item ?y - item): ?x is unstacked from ?y. This predicate applies when ?x
 and ?y are blocks.

**Task Environment:**
In the environment where the demonstrations are being performed, there are the following
objects:
- A table. Objects can be placed on the table.
- A drawer that can be opened. Objects can be placed into the drawer when it is open.
- A slider which is a cabinet with a sliding door. The sliding door can be moved to the
left or to the right. Objects can be placed into the slider no matter the position of the
sliding door.
- A lightbulb that be can turned on/off with a button.
- A led that can be turned on/off with a button.
- Three blocks that can be rotated, pushed, lifted, and placed.
```

Listing L.4: **Example Prompt for CALVIN–In-Context Example.**

```
**Demonstration Parsing:**
Now, you will help to parse several human demonstrations of the robot performing a task
and generate a lifted description of how to accomplish this task.
For each demonstration, a sequence of performed primitives will be given, with actual
object names. Three demonstrations for the task of "place_in_slider" is:

<code name="primitive_sequence">
primitives = [
  {"name": "grasp", "arguments": ["red_block", "table"]}
  {"name": "move", "arguments": ["red_block"]}
  {"name": "place", "arguments": ["red_block", "slider"]}
```

```
  {"name": "move-to", "arguments": [""]}
]
</code>

<code name="primitive_sequence">
primitives = [
  {"name": "grasp", "arguments": ["blue_block", "table"]}
  {"name": "move", "arguments": ["blue_block"]}
  {"name": "place", "arguments": ["blue_block", "slider"]}
  {"name": "move-to", "arguments": [""]}
]
</code>

<code name="primitive_sequence">
primitives = [
  {"name": "grasp", "arguments": ["pink_block", "table"]}
  {"name": "move", "arguments": ["pink_block"]}
  {"name": "place", "arguments": ["pink_block", "slider"]}
  {"name": "move-to", "arguments": [""]}
]
</code>

**Previous Tasks:**
A list of tasks that can be performed before the current task will also be provided as
context. For the task of "place_in_slider", the possible previous tasks are:
lift_block_table, lift_block_drawer, move_slider_right

**Example Output:**
You should generate a lifted description, treating all objects as variables. For example,
the lifted description for "place_in_slider" is:
<code name="mechanism">
(:mechanism place-in-slider
 :parameters (?block - item ?slider - item)
 :precondition (and (is-block ?block) (is-slider ?slider) (is-lifted ?block))
 :effect (and (is-in ?block ?slider) (not (is-lifted ?block)))
 :body (then
   (place ?block ?slider)
 )
)
</code>
```

Listing L.5: **Example Prompt for CALVIN–Instructions.**

```
**Think Step-by-Step:**
To generate the lifted description, you should think through the task in natural language
in the following steps. Be EXTREMELY CAREFUL to think through step 3a, 3b, and 4a, 4b.
1. Parse the goal. For example "place_in_slider", the goal is to place a block into the
slider.
2. Think about the possible effects achieved by previous tasks and the previous actions
that have been performed. For "lift_block_table", a block is lifted from the table and the
 effect is that the block is lifted. For "lift_block_drawer", a block is lifted from the
drawer and the effect is that the block is lifted. For "move_slider_right", the sliding
door of the slider is moved to the right and the effect is that the sliding door is on the
 right.
3. Parse the demonstrations and choose the combination of primitives for the current task.
 The demonstrations are noisy so that the demonstrated primitive sequences may include
```

```
extra primitive actions that are not necessary for the current task at the beginning or
end. The extra primitive actions can be for the previous tasks. Combining with the
understanding of the task and previous task to infer the correct combination of primitives
 for the current task.
3a. In this case, the previous tasks are relevant to the current task. We should think
about how to sequence the previous tasks with the current task. The primitive combination
for the current task should not include primitive actions that have been performed. The
above example for "place_in_slider" is this case. We can infer that "grasp" in the
demonstrated sequences is likely to be for the previous tasks and should not be included
in the primitive combination for the current task. We therefore choose B. (then (place ?x
?y)). The semantics is that the robot place the lifted block in the slider.
3b. In this case, the previous tasks are not relevant to the current task.
4. Think about the preconditions. Also specify the types of all relevant objects in the
preconditions.
4a. In this case, previous tasks are relevant to the current task. We should think about
the effects of the previous tasks. For "place_in_slider", the effects of previous tasks
include the block is already lifted. So we should specify that the block is lifted in the
preconditions for the current task.
4b. In this case, previous tasks are not relevant to the current task.
5. Think about the effects. For "place_in_slider", the effects are that the block is in
the slider and the block is not lifted.
6. Write down the mechanism in the format of the example.

**Additional Instructions:**
1. Make sure the generated lifted description starts with <code name="mechanism"> and ends
 with </code>.
2. Please do not invent any new predicates for the precondition and effect. You can only
use the predicates listed above.
3. Consider the physical constraints of the objects. For example, a robot arm can not go
through a closed door.
4. For each parameter in :parameters, you should use one of the predicates for specifying
the type of the object to indicate its type (e.g., is-drawer, is-block, and etc).
```

Listing L.6: **Example Prompt for CALVIN–Task Input.**

```
**Current Task:** place_in_drawer

**Example Sequences:**
<code name="primitive_sequence">
primitives = [
  {"name": "grasp", "arguments": ["blue_block", "table"]}
  {"name": "move", "arguments": ["blue_block"]}
  {"name": "place", "arguments": ["blue_block", "drawer"]}
  {"name": "move-to", "arguments": [""]}
]
</code>

<code name="primitive_sequence">
primitives = [
  {"name": "grasp", "arguments": ["red_block", "table"]}
  {"name": "move", "arguments": ["red_block"]}
  {"name": "place", "arguments": ["red_block", "drawer"]}
  {"name": "move-to", "arguments": [""]}
]
</code>
```

```
<code name="primitive_sequence">
primitives = [
  {"name": "grasp", "arguments": ["pink_block", "table"]}
  {"name": "move", "arguments": ["pink_block"]}
  {"name": "place", "arguments": ["pink_block", "drawer"]}
  {"name": "move-to", "arguments": [""]}
]
</code>

**Previous Tasks:** push_into_drawer, lift_block_table, lift_block_slider
```

Listing L.7: **Example Prompt for Predicate Generation.**

```
You are a helpful agent in helping a robot interpret human demonstrations and discover a
generalized high-level routine to accomplish a given task.
**Primitive Actions:**
There are seven primitive actions that the robot can perform. They are:
- (grasp ?x ?y): ?x and ?y are two object variables. ?x is the object that the robot will
be grasping, ?y is the object that ?x is currently on or in.
- (place ?x ?y): ?x and ?y are two object variables. ?x is the object that the robot is
currently holding, ?y is the object that ?x will be placed on or in.
- (move ?x): ?x is the object that the robot is currently holding and will be moved by the
 robot.
- (push ?x): ?x is the object that the robot will be pushing.
- (move-to ?x): the robot arm will move without holding any object or pushing any object.
- (open): the robot gripper will open fully.
- (close): the robot gripper will close without grasping any object.

**Task Environment:**
In the environment where the demonstrations are being performed, there are the following
objects:
- A table. Objects can be placed on the table.
- A drawer that can be opened. Objects can be placed into the drawer when it is open.
- A slider which is a cabinet with a sliding door. The sliding door can be moved to the
left or to the right. Objects can be placed into the slider no matter the position of the
sliding door.
- A lightbulb that be can turned on/off with a button.
- A led that can be turned on/off with a button.
- Three blocks that can be rotated, pushed, lifted, and placed.

**Task**
You will help the robot to write PDDL definitions for the following actions:
1. lift_red_block_table
2. lift_red_block_slider
3. lift_red_block_drawer
4. lift_blue_block_table
5. lift_blue_block_slider
6. lift_blue_block_drawer
7. lift_pink_block_table
8. lift_pink_block_slider
9. lift_pink_block_drawer
10. stack_block
11. unstack_block
12. place_in_slider
13. place_in_drawer
14. place_on_table
```

```
15. rotate_red_block_right
16. rotate_red_block_left
17. rotate_blue_block_right
18. rotate_blue_block_left
19. rotate_pink_block_right
20. rotate_pink_block_left
21. push_red_block_right
22. push_red_block_left
23. push_blue_block_right
24. push_blue_block_left
25. push_pink_block_right
26. push_pink_block_left
27. move_slider_left
28. move_slider_right
29. open_drawer
30. close_drawer
31. turn_on_lightbulb
32. turn_off_lightbulb
33. turn_on_led
34. turn_off_led

Before writing the operators, define the predicates that should be used to write the
preconditions and effects of the operators. Group the predicates into unary predicates
that define the states of objects and binary relations that specify relations between two
objects. For each predicate, list actions that are relevant.
```

Listing L.8: **Prompt for SayCan.**

```
**Objective:**
You are a helpful agent in helping a robot plan a sequence of actions to accomplish a
given task.
I will first provide context and then provide an example of how to perform the task.

**Task Environment:**
In the robot's environment, there are the following objects:
- A table. Objects can be placed on the table.
- A drawer that can be opened. Objects can be placed into the drawer when it is open.
- A slider which is a cabinet with a sliding door. The sliding door can be moved to the
left or to the right. Objects can be placed into the slider no matter the position of the
sliding door.
- A lightbulb that be can turned on/off with a button.
- A led that can be turned on/off with a button.
- Three blocks that can be rotated, pushed, lifted, and placed.

**Actions:**
There are the following actions that the robot can perform. They are:
- lift_red_block_table: lift the red block from the table.
- lift_red_block_slider: lift the red block from the slider.
- lift_red_block_drawer: lift the red block from the drawer.
- lift_blue_block_table: lift the blue block from the table.
- lift_blue_block_slider: lift the blue block from the slider.
- lift_blue_block_drawer: lift the blue block from the drawer.
- lift_pink_block_table: lift the pink block from the table.
- lift_pink_block_slider: lift the pink block from the slider.
- lift_pink_block_drawer: lift the pink block from the drawer.
- stack_block: stack the blocks.
```

```
- place_in_slider: place the block in the slider.
- place_in_drawer: place the block in the drawer.
- place_on_table: place the block on the table.
- rotate_red_block_right: rotate the red block to the right.
- rotate_red_block_left: rotate the red block to the left.
- rotate_blue_block_right: rotate the blue block to the right.
- rotate_blue_block_left: rotate the blue block to the left.
- rotate_pink_block_right: rotate the pink block to the right.
- rotate_pink_block_left: rotate the pink block to the left.
- push_red_block_right: push the red block to the right.
- push_red_block_left: push the red block to the left.
- push_blue_block_right: push the blue block to the right.
- push_blue_block_left: push the blue block to the left.
- push_pink_block_right: push the pink block to the right.
- push_pink_block_left: push the pink block to the left.
- move_slider_left: move the slider to the left.
- move_slider_right: move the slider to the right.
- open_drawer: open the drawer.
- close_drawer: close the drawer.
- turn_on_lightbulb: turn on the lightbulb.
- turn_off_lightbulb: turn off the lightbulb.
- turn_on_led: turn on the led.
- turn_off_led: turn off the led.
- do_nothing: do nothing.

**Example Task:**
Now, you will help to parse the goal predicate and generate a list of candidate actions
the robot can potentially take to accomplish the task. You should rank the actions in
terms of how likely they are to be performed next.
Goal predicate: (is-turned-off led)
Task output:
```python
['turn_off_led', 'do_nothing']
```
In this example above, if the led is on, the robot should turn it off. If the led is
already off, the robot should do nothing.

When the robot successfully completes an action, the robot will ask for the next action to
 take.
Considering the executed task: turn_off_led
Your output should be:
```python
['do_nothing']
```
Since the led is already off, the robot should do nothing.

Additional Instructions:
1. Make sure the generated plan is a list of actions. Place the list between ```python and
 ends with ```.
2. Think Step-by-Step.

Goal predicate: {Based on the given task}
Current symbolic state: {Based on the simulator state}
Executed actions: {Based on the previously executed actions}
```

Listing L.9: **Initial Prompt for Robot-VILA.**

```
You are highly skilled in robotic task planning, breaking down intricate and long-term
tasks into distinct primitive actions.
If the object is in sight, you need to directly manipulate it. If the object is not in
sight, you need to use primitive skills to find the object
first. If the target object is blocked by other objects, you need to remove all the
blocking objects before picking up the target object. At
the same time, you need to ignore distracters that are not related to the task. And
remember your last step plan needs to be "done".

Consider the following skills a robotic arm can perform.
- lift_red_block_table: lift the red block from the table.
- lift_red_block_slider: lift the red block from the slider.
- lift_red_block_drawer: lift the red block from the drawer.
- lift_blue_block_table: lift the blue block from the table.
- lift_blue_block_slider: lift the blue block from the slider.
- lift_blue_block_drawer: lift the blue block from the drawer.
- lift_pink_block_table: lift the pink block from the table.
- lift_pink_block_slider: lift the pink block from the slider.
- lift_pink_block_drawer: lift the pink block from the drawer.
- stack_block: stack the blocks.
- place_in_slider: place the block in the slider.
- place_in_drawer: place the block in the drawer.
- place_on_table: place the block on the table.
- rotate_red_block_right: rotate the red block to the right.
- rotate_red_block_left: rotate the red block to the left.
- rotate_blue_block_right: rotate the blue block to the right.
- rotate_blue_block_left: rotate the blue block to the left.
- rotate_pink_block_right: rotate the pink block to the right.
- rotate_pink_block_left: rotate the pink block to the left.
- push_red_block_right: push the red block to the right.
- push_red_block_left: push the red block to the left.
- push_blue_block_right: push the blue block to the right.
- push_blue_block_left: push the blue block to the left.
- push_pink_block_right: push the pink block to the right.
- push_pink_block_left: push the pink block to the left.
- move_slider_left: move the slider to the left.
- move_slider_right: move the slider to the right.
- open_drawer: open the drawer.
- close_drawer: close the drawer.
- turn_on_lightbulb: turn on the lightbulb.
- turn_off_lightbulb: turn off the lightbulb.
- turn_on_led: turn on the led.
- turn_off_led: turn off the led.
- done: the goal has reached.

You are only allowed to use the provided skills. You can first itemize the task-related
objects to help you plan.
For the actions you choose, list them as a list in the following format.

<code>
['turn_off_led', 'open_drawer', 'done']
</code>
```

Listing L.10: **Follow-Up Prompt for Robot-VILA.**

```
This image displays a scenario after you have executed some steps from the plan generated
earlier. When interacting with people, sometimes the robotic arm needs to wait for the
person's action. If you do not find the target object in the current image, you need to
continue searching elsewhere. Continue to generate the plan given the updated environment
state.
```

Listing L.11: **Prompt for Text2Motion.**

```
**Objective:**
You are a helpful agent in helping a robot plan a sequence of actions to accomplish a
given task.
I will first provide context and then provide an example of how to perform the task.

**Task Environment:**
In the robot's environment, there are the following objects:
- A table. Objects can be placed on the table.
- A drawer that can be opened. Objects can be placed into the drawer when it is open.
- A slider which is a cabinet with a sliding door. The sliding door can be moved to the
left or to the right. Objects can be placed into the slider no matter the position of
the sliding door.
- A lightbulb that be can turned on/off with a button.
- A led that can be turned on/off with a button.
- Three blocks that can be rotated, pushed, lifted, and placed.

**Predicates for symbolic state:**
The list of all possible predicates for defining the symbolic state are listed below:
- (rotated-left ?x - item): ?x is rotated left. This predicate applies to a block.
- (rotated-right ?x - item): ?x is rotated right. This predicate applies to a block.
- (pushed-left ?x - item): ?x is pushed left. This predicate applies to a block.
- (pushed-right ?x - item): ?x is pushed right. This predicate applies to a block.
- (lifted ?x - item): ?x is lifted in the air. This predicate applies to a block.
- (is-open ?x - item): ?x is open. This predicate applies to a drawer.
- (is-close ?x - item): ?x is close. This predicate applies to a drawer.
- (is-turned-on ?x - item): ?x is turned on. This predicate applies to a lightbulb or a
led.
- (is-turned-off ?x - item): ?x is turned off. This predicate applies to a lightbulb or
a led.
- (is-slider-left ?x - item): the sliding door of the slider ?x is on the left.
- (is-slider-right ?x - item): the sliding door of the slider ?x is on the right.
- (is-on ?x - item ?y - item): ?x is on top of ?y. This predicate applies when ?x is a
block and ?y is a table.
- (is-in ?x - item ?y - item): ?x is inside of ?y. This predicate applies when ?x is a
block and ?y is a drawer or a slider.
- (stacked ?x - item ?y - item): ?x is stacked on top of ?y. This predicate applies when
 ?x and ?y are blocks.
- (unstacked ?x - item ?y - item): ?x is unstacked from ?y. This predicate applies when
?x and ?y are blocks.

**Actions:**
There are the following actions that the robot can perform. They are:
- lift_red_block_table: lift the red block from the table.
- lift_red_block_slider: lift the red block from the slider.
- lift_red_block_drawer: lift the red block from the drawer.
- lift_blue_block_table: lift the blue block from the table.
- lift_blue_block_slider: lift the blue block from the slider.
- lift_blue_block_drawer: lift the blue block from the drawer.
```

- lift_pink_block_table: lift the pink block from the table.
- lift_pink_block_slider: lift the pink block from the slider.
- lift_pink_block_drawer: lift the pink block from the drawer.
- stack_block: stack the blocks.
- place_in_slider: place the block in the slider.
- place_in_drawer: place the block in the drawer.
- place_on_table: place the block on the table.
- rotate_red_block_right: rotate the red block to the right.
- rotate_red_block_left: rotate the red block to the left.
- rotate_blue_block_right: rotate the blue block to the right.
- rotate_blue_block_left: rotate the blue block to the left.
- rotate_pink_block_right: rotate the pink block to the right.
- rotate_pink_block_left: rotate the pink block to the left.
- push_red_block_right: push the red block to the right.
- push_red_block_left: push the red block to the left.
- push_blue_block_right: push the blue block to the right.
- push_blue_block_left: push the blue block to the left.
- push_pink_block_right: push the pink block to the right.
- push_pink_block_left: push the pink block to the left.
- move_slider_left: move the slider to the left.
- move_slider_right: move the slider to the right.
- open_drawer: open the drawer.
- close_drawer: close the drawer.
- turn_on_lightbulb: turn on the lightbulb.
- turn_off_lightbulb: turn off the lightbulb.
- turn_on_led: turn on the led.
- turn_off_led: turn off the led.

Example Task:
Now, you will help to parse the goal predicate and generate a sequence of actions to accomplish this task.
Goal predicate: (is-turned-off led)
Symbolic state: is-turned-on(led), is-turned-on(lightbulb), not(is-turned-off(led)), not(is-turned-off(lightbulb))
Task output:
```python
['turn_off_led']
```

Example Task:
Goal predicate: (is-turned-on led)
Symbolic state: is-turned-on(led), is-turned-on(lightbulb), not(is-turned-off(led)), not(is-turned-off(lightbulb))
Task output:
```python
[]
```

Example Task:
Goal predicate: (is-in red_block drawer)
Symbolic state: not(is-in(red_block, drawer)), not(is-in(red_block, slider)), is-on(red_block, table), not(is-open(drawer)), is-close(drawer), is-slider-left(slider), not(is-slider-right(slider)), not(lifted(red_block))
Task output:
```python
['open_drawer', 'lift_red_block_table', 'place_in_drawer']
```

Example Task:
Goal predicate: (is-in red_block drawer)
Symbolic state: not(is-in(red_block, drawer)), not(is-in(red_block, slider)), not(is-on(red_block, table)), is-open(drawer), not(is-close(drawer)), is-slider-left(slider), not(is-slider-right(slider)), lifted(red_block)
Task output:
```python
['place_in_drawer']
```

Example Task:
Goal predicate: (and (is-turned-on lightbulb) (is-slider-right slider))
Symbolic state: is-slider-left(slider), not(is-slider-right(slider)), is-turned-off(lightbulb), not(is-turned-on(lightbulb))
Task output:
```python
['turn_on_lightbulb', 'move_slider_right']
```

Additional Instructions:
1. Make sure the generated plan is a list of actions. Place the list between ```python and ends with ```.
2. Think Step-by-Step.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix M

Appendix for Hybrid Imperative-Declarative Action Representations

M.1 Refinements and The CROW Interpreter

In this section, we formally define the refinement rules for CDL behavior rules. We start with a definition of types of statement orders.

Statement orders. For a set of statements, we define two basic orders.

1. Sequential program, represented as an ordered set $\{p_1, p_2, \cdots, p_k\}_{seq}$. Intuitively, all statements will be executed in this order.
2. Unordered program, represented as an unordered set $\{p_1, p_2, \cdots, p_k\}_{unordered}$. Intuitively, all statements can be executed in arbitrary orders (there are in total $k!$ possible serialization of the program).

These two types of primitive ordering can be nested, to represent partial orders. For example, the ordering $\{\{p_1, p_2\}_{seq}, \{p_3, p_4\}_{seq}\}_{unordered}$ states that p_1 will be executed before p_2 and that p_3 will be executed before p_4. However, the relative ordering between, for example, p_1 and p_3 is not specified. In general, we will call these programs *partially ordered programs*. The set of all possible partially ordered plans (including sequential ones and unordered ones) is denoted as $\mathcal{P}$.

As an example, consider the following `foreach` statement.

```
1  foreach o: T:
2    statement_1(o); statement_2(o); ...
```

In a domain with two objects, it would be effectively equivalent to

```
unordered:
  sequential:
    statement_1(o1); statement_2(o1); ...
  sequential:
    statement_1(o2); statement_2(o2); ...
```

Next, we define the *tail* operator that takes a partially ordered program p and returns a set of statements in p that might be the "last" statement in one of the serializations of p. It can be defined recursively:

$$tail(p) = \begin{cases} \{p\} & \text{if } p \text{ is a single subgoal or a controller call} \\ tail(p_k) & \text{if } p = \{p_1, \cdots, p_k\}_{seq} \text{ is a totally ordered plan} \\ \bigcup_{p_i \in p} tail(p_i) & \text{if } p \text{ is an unordered plan} \end{cases}.$$

One can iteratively select an element from $tail(p)$ nondeterministically to construct all possible serializations of p. In CDL, there are two keywords `sequential` and `unordered` for specifying a particular section of the program being sequential or unordered. In CDL, a behavior body can contain unordered sections of `achieve` statements, indicating a scenario where the order for achieving those subgoals is unspecified by the programmer, but rather, should be determined using search.

The generalized refinement operator. Each behavior body is a program $P = \langle L, M, R \rangle$ with three sections: left (everything before the promotable section), middle (the promotable section), and right (everything after the promotable section). The generalized plan refinement operator $\mathcal{R}^* : \mathcal{P}^3 \to \mathcal{P}_0$ takes as input a program tuple $P = \langle L, M, R \rangle$ and nondeterministically refines it into a plan trace $\bar{a}$. $\mathcal{R}^*(P)$ iteratively replaces one of the `achieve` statements in P with the body of a behavior b whose goal matches the achieve statement.

Next, to handle possible orders of achieving subgoals, it non-deterministically selects a statement β from $tail(L), tail(M)$, or $tail(R)$. If β is a non-achieve statement, it is directly appended to the plan. Otherwise, we non-deterministically select a behavior b that matches β to handle multiple behavior rules for achieving the goal β.

Finally, to handle interleaved execution, we consider two cases: when β is from M and when β is not. Let $\langle L', M', R' \rangle$ be the three sections of $body(b)$. When β is not from M, we simply refine $\langle L', M', R' \rangle$ recursively and stitches together the outputs. When β is from M. Informally, statements in L' will be appended to L, R' will be prepended to R, and M will be mixed with the rest part of M (i.e., $M \setminus \{\beta\}$, where the $\setminus$ operator (set difference) returns a partially ordered plan just as M but with β removed). The following program shows the

program that defines REFINE operator algorithmicly.

function REFINE($P = \langle L, M, R \rangle$)
  for β nondeterministically from $tail(L) \cup tail(M) \cup tail(R)$ **do**
    if $\beta \in tail(L)$ **then**
      for $\bar{a}_1 \in$ REFINE$(L \setminus \{\beta\}, \emptyset, \emptyset)$, $\bar{a}_2 \in$ REFINE$(\emptyset, M, R)$ **do**
        if β is `achieve` **then**
          for $b \in \mathcal{B} \mid goal(b) = \beta$ **do yield** $\bar{a}_1 \oplus$ REFINE$(b) \oplus \bar{a}_2$
        else yield $\bar{a}_1 \oplus \{\beta\}_{seq} \oplus \bar{a}_2$
    else if $\beta \in tail(R)$ **then** ▷ Similarly to the case of $\beta \in tail(L)$
      for $\bar{a}_1 \in$ REFINE$(L, M, \emptyset)$, $\bar{a}_2 \in$ REFINE$(\emptyset, \emptyset R \setminus \{\beta\})$ **do**
        if β is `achieve` **then**
          for $b \in \mathcal{B} \mid goal(b) = \beta$ **do yield** $\bar{a}_1 \oplus$ REFINE$(b) \oplus \bar{a}_2$
        else yield $\bar{a}_1 \oplus \{\beta\}_{seq} \oplus \bar{a}_2$
    else if $\beta \in tail(M)$ **then**
      if β is `achieve` **then**
        for $b = \langle L', M', R' \rangle \in \mathcal{B} \mid goal(b) = \beta$ **do**
          yield REFINE$(\{L, L'\}_{seq}, \{M \setminus \{\beta\}, M'\}_{unordered}, \{R', R\}_{seq})$
      else
        for $\bar{a}_1 \in$ REFINE$(L, M \setminus \{\beta\}, \emptyset)$, $\bar{a}_2 \in$ REFINE$(\emptyset, \emptyset, R)$ **do**
          yield $\bar{a}_1 \oplus \{\beta\}_{seq} \oplus \bar{a}_2$

It is important to observe that CDL separates the definition of the generative model for plan traces in the behavior rule bodies from the definition of preconditions and effects of taking actions in the world in the assertions and effects clauses of behavior rules. This allows the behavior specification to be separated into two models: $\mathcal{G}$, a possible plan generator, and $\mathcal{T}$, a plan soundness tester. The task of a CDL interpreter is to find a plan $P \in \mathcal{G}$ such that $\mathcal{T}(P)$. The most naïve implementation would be to enumerate $P \in \mathcal{G}$ in some order and return the first plan P where $\mathcal{T}(P)$ is true. This method would generally be very inefficient, or even incomplete in some continuous domains. In section 12.3 we provide an algorithm that leverages soundness checks on partially refined plan traces, as well as other mechanisms to dramatically improve efficiency. A more advanced algorithm may leverage partially refined plans from $\mathcal{G}$ (recall that the plan is generated hierarchically), and run tests on part of the plan, to be more efficient. Other methods such as leveraging heuristics or optimizing expansion ordering are also compatible.

M.1.1 The Completeness and Soundness of Crow

Based on the algorithmic definition of refinement operators, we can now prove the soundness of the Crow interpreter, as stated in Theorem 12.3.1.

Proof. The high-level idea is to prove that, under a given depth, Crow will enumerate all possible plan traces that can be generated by the Refine operator. Then, since we are iteratively increasing the allowed depth for the algorithm it is guaranteed that a sound plan will be generated.

We now prove the completeness of the enumerate strategy that Crow employs. Given a particular program $\langle L, M, R\rangle$, we show that $\textsc{Refine}(\langle L, M, R\rangle)$ can be decomposed into the refinement of three programs L, M, and R independently. Formally:

$$\begin{aligned}\textsc{Refine}(\langle L, M, R\rangle) = \{\bar{a}_1 \oplus \bar{a}_2 \oplus \bar{a}_3 \mid & \bar{a}_1 \in \textsc{Refine}(\langle L, \emptyset, \emptyset\rangle), \\ & \bar{a}_2 \in \textsc{Refine}(\langle \emptyset, M, \emptyset\rangle), \bar{a}_3 \in \textsc{Refine}(\langle \emptyset, \emptyset, R\rangle)\}\end{aligned}$$

For convenience, we define

$$\textsc{Refine}(P) \times \textsc{Refine}(Q) = \{\bar{a}_1 \oplus \bar{a}_2 \mid \bar{a}_1 \in \textsc{Refine}(P), \bar{a}_2 \in \textsc{Refine}(Q)\}$$

This can be proved by induction. First, when $L = M = R = \emptyset$ this is correct. Now consider for any $\beta \in \mathit{tail}(L) \cup \mathit{tail}(M) \cup \mathit{tail}(R)$:

1. If $\beta \in \mathit{tail}(L)$, the refinement is already explicitly decomposed into the refinement of $\textsc{Refine}(\langle L, \emptyset, \emptyset\rangle) \times \textsc{Refine}(\langle \emptyset, M, R\rangle)$, which, by induction is equal to $\textsc{Refine}(\langle L, \emptyset, \emptyset\rangle) \times \textsc{Refine}(\langle \emptyset, M, \emptyset\rangle) \times \textsc{Refine}(\langle \emptyset, \emptyset, R\rangle)$.

2. If $\beta \in \mathit{tail}(R)$, the refinement is symmetric to the L case.

3. If $\beta \in \mathit{tail}(M)$, we are recursively refining the following program: $\langle \{L, L'\}_{seq}, \{M \setminus \{\beta\}, M'\}_{unordered}, \{R', R\}_{seq}\rangle$. For a sequential program $\{L, L'\}_{seq}$, its refinement has the property that $\textsc{Refine}(\{L, L'\}_{seq}) = \textsc{Refine}(L) \times \textsc{Refine}(L')$ by the definition of *tail*. Therefore, this refinement is equivalent to: $\textsc{Refine}(L) \times \textsc{Refine}(L') \times \textsc{Refine}(\{M \setminus \{\beta\}, M'\}_{unordered}) \times \textsc{Refine}(R') \times \textsc{Refine}(R) = \textsc{Refine}(\langle L, \emptyset, \emptyset\rangle) \times \textsc{Refine}(\langle \emptyset, M, \emptyset\rangle) \times \textsc{Refine}(\langle \emptyset, \emptyset, R\rangle)$

Therefore, the Crow left-most refinement strategy is complete. And it is sound because it all assertions have been explicitly tested either during the refinement process, or during the CSP solving. □

M.2 Formulations and Algorithms for Constraint Satisfaction Problems

A constraint satisfaction problem is a tuple of $\langle V, C \langle$, where V is the set of variables and C is a set of constraints. CSP solving focuses on finding a satisficing solution to all variables in V that satisfies all constraints in C. Each constraint $c \in C$ is a function that takes variables in V as inputs and returns a Boolean value.

The caveat of using the CSP formulation in decision-making is that the constraints are defined alongside the plan refinement processes. Therefore, depending on the particular time t when specific constraints are evaluated and added to the constraint set, they will have the state s_t implicitly as inputs to those constraints functions. In this case, these functions are no-longer pure functions in the sense that they depend on additional information other than its input.

There are multiple ways to handle this. In our implementation, we handle this using situation calculus. In particular, each feature $f(...)$ should now be considered as a fluent $f_t(...)$. For example, consider we have the following assignment statement at step t of a feature $f(o_1, o_2)$.

```
1  f[o_1, o_2] = g(arg1, arg2)
```

In this case, if `arg1` or `arg2` are free variables, the evaluation result of the function g will also be a free variable (e.g., `f_t_o1_o2`) which satisfies the following constraint:

```
1  assert f_t_o1_o2 = g(arg1, arg2)
```

We describe such translations in detail in Appendix M.5.5.

Using this translation, all we have is a set collection of constraints that are pure, which can be solved by a CSP solver.

M.2.1 Sampling-based Continuous Constraint Satisfaction Problem Solver

In Crow, we implemented a solver based on sampling from generators following Garrett *et al.* [2018]. It is a depth-first-search-based solver using the DPLL heuristic for acceleration. At a high level, at each step, it first searches for variables whose values can be fully determined based on the current set of assignments A. This includes three cases:

1. The constraint is Boolean constraint, such as `a and b and c and ...`. Then we immediately know that all variables `a, b, c, ...` must be true. If it is `a or b or c or ...` and all variables except for `a` has been assigned to false, then we know `a` must be true.

2. The constraint is an equality constraint of type `f(...) = rv` and all the arguments to the function `f` have been determined. In this case, we can deterministically evaluate `f` to get the assignment to `rv`.

3. There is an equality constraint `f(...) = rv` but the return value `rv` is never appeared in other constraints except for this equality constraint. We can ignore this constraint (and `rv`).

If none of these cases are true, we consider performing a search over the free variables. We prioritize the search for discrete variables (e.g., Boolean values). We essentially non-deterministically choose an assignment to the variable and continue the search for other variables, and backtrack if we can not find assignments to other variables.

When there is no remaining discrete variables, we consider sampling for continuous variables (e.g., grasps, poses, and trajectories). In this case, we will iterate over all constraints and find constraints that matches a user-provided generator. We will describe the detailed syntax for generators in Appendix M.5 but in short, a generator is defined using the following syntax.

```
generator gen_f(x: T, y: T, ...):
  goal: f(x, ...)= y
  in: y, ...
  out: x
```

Here we have defined a generator associated with the function `f`. Given the value of `y` and other arguments to the function `f`, this generator will generate a set of values for the variable `x` such that `f(x, ...) = y` evaluates to true. If we locate a constraint that matches the input-output specification of the generator, we will try to apply this generator to find the value for the variable `x`. In practice, since there may be an infinite number of values satisfying a certain constraint, we will always put an upper bound on the number of values that we consider for each generator.

M.3 Proofs for Planner Complexity

The planner Crow can be inefficient when it indeed needs to nondeterministically enumerate all possible plans. It is useful to introduce a special keyword `commit` to restrict the amount of branching it performs. In short, `commit` is a primitive statement that can be used in the right section of the behaviors. It states that for all statements on its "left", it only needs to consider a single plan for achieving those subgoals. Specifically, let $\beta = tail(R)$ of the program $\langle L, M, R \rangle$, and the function returns only a single controller sequence and a single

assignment to all continuous variables. That is, for all s, for all tuples of $\langle s', \bar{a}, V', C' \rangle$ generated by CROW$(s, \langle L, M, R \setminus \{\beta\}\rangle)$, it only returns a single tuple of $\langle s'\bar{a}, \emptyset, \emptyset$ where all constraints in $\langle V', C' \rangle$ has solved and committed.

Next, let's consider the set of behavior programs where all statements are committed immediately afterward. We additionally define two notations. For a given state s, let $Opt(s, P)$ denote the set of all optimal primitive plans (in terms of the number of primitive controller calls) that is in $\mathcal{R}^*(P)$ and is sound given initial state s. Second, let $Exec(s, P)$ be the state after applying all effect statements in a primitive plan P starting from state s.

For a given state s_0, a partially ordered plan P is called *strong serialization solvable* if and only if there is an element $q \in tail(P)$* and behavior b where $goal(b) = q$ and $body(b) = \langle L, M, R \rangle$, such that:

1. $\{P \setminus \{q\}, M\}_{unordered}$ is strong serialization solvable.
2. for all optimal primitive plans $P_1 \in Opt(s_0, \{P \setminus \{q\}, M\}_{unordered})$, and for all optimal primitive plans $P_2 \in Opt(Exec(s_0, P_1), R)$, $P_1 \oplus P_2 \in Opt(s_0, P)$.

This property intuitively states that, in order to find a sound primitive plan for P, we can independently solve $\{P \setminus \{q\}, M\}_{unordered}$, and then solve for R based on the resulting sequence of primitive controller calls (and behavior effects). Note that this is a slightly different notation than the case where one may define $Opt(s, P)$ as the oracle "optimal" plan because here we only consider plans that can be generated by our $\mathcal{R}^*$ operator, which is conditioned on the set of behavior specifications we have.

Theorem M.3.1 (Regression Width, adapted from Chapter 4)**.** *Consider only planning problems without continuous variables (therefore, there is no* `bind` *statements). Suppose all behaviors do not contain for-loops. Let K be the maximum number of* `achieve` *statements in a program* $\langle L, M, R \rangle$. CROW *will run in* $N^{O(K)}$ *where N is the number of entities in the environment, and K, also called the "width" of the planning problem, is the maximum number of statements in the set M during search.*

Proof sketch. The key observation behind the proof is that for all possible programs p, let $subgoals(p)$ be the set of `achieve` statements inside p. No matter the order of elements in p, all we care about is to find an (optimal) plan that simultaneously achieves all subgoals in $subgoals(p)$, and there is only $N^{O(K)}$ possible subsets.

This gives us a fine-grained insight into a useful class of planning problems in terms of their hardness. Essentially, considering discrete planning problems, the hardness of the problem essentially depends on the total number of subgoals that accumulated in the program, especially the "middle" section. In order to analyze how this property gets accumulated in

*we only consider achievement statements here. For other types of statements q this property can be simply defined because they do not require further refinement.

a practical problem, we can consider individual operators separately, and then add up the number of subgoals in a search process. Of course, the hardness would depend on the initial state and the goal.

Importantly, this property of width also naturally extends to the solving of continuous constraint satisfaction. In particular, let us define $Opt^{cont}(s, P, V)$ as the set of optimal primitive plans with variables V that is in $\mathcal{R}^*(P)$ with all variables in V already grounded. The same definition of *strong serialization solvable directly generalizes*. Intuitively, this extended property states "commitments" not only for discrete action choices but also for the choice of continuous parameters—all `bind` statements in the promotable section need to be jointly considered with other objects. If we hypothesize that the amount of time needed for solving a set of continuous constraints is exponential in terms of the number of variables, then this gives us an estimate of both the complexity of the CROW as well as the fundamental hardness of a planning problem.

M.4 Implementation Details for NAMO

Our most advanced strategy (interleaved-waypoint-connect) is implemented using the following CDL programs:

```
behavior NAMO(g: Pose):
goal: at(g)
body:
  bind x: Pose:
    valid_waypoint(x)
  bind t: Trajectory where:
    connect(x, g, t)
  promotable:
    achieve at(x)
    forall o: Object:
      achieve not blocking(o, t)
  do exec_traj(t)
eff:
  agent_position = g

# The base case of NAMO for the first waypoint.
behavior NAMO_base(g: Pose):
goal: at(g)
body:
  bind t: Trajectory where:
```

```
    connect(agent_position, g, t)
  promotable:
    forall o: Object:
      achieve not blocking(o, t)
  do exec_traj(t)
eff:
  agent_position = g

behavior move_away(o: Object, t: Trajectory)
goal: not blocking(o, t)
body:
  let original_pos = agent_position
  # Move the agent to a position where it is close to o
  achieve close_to(o)
  do attach(o)
  # Find a free location for the obstacle o away from t
  bind o_pos = free_location(o, t, o_pos)
  achieve position(o) == o_pos
  do detach(o)
  # Move back to the previous position
  bind t: collision_free_connect(agent_position, original_pos, t)
  do exec_traj(t)

behavior approach(o: Object):
goal: close_to(o)
body:
  # Find a position for the agent that is close to the object
  bind x: Pose where:
    close_to_object(o, x)
  # Find a trajectory from the agent's current position
  #   to the target position
  bind t: Trajectory where:
    collision_free_connect(agent_position, x, t)
  do exec_traj(t)
eff:
  agent_position = x

behavior move_object_to(o: Object, p: Pose):
goal: position(o) == p
body:
```

```
  # Find a trajectory for moving the object to the target position
  bind t: Trajectory where:
    collision_free_connect_with_object(position(o), p, t)
  do exec_traj(t)
```

We implement the following basic generators for our NAMO domain.

1. `valid_waypoint`: we randomly sample a point that is not directly reachable from the initial state of the agent. Otherwise, it should be handled by the base case.

2. `connect`: we randomly sample a path (using A* search on the grid) that is not colliding with walls (i.e., non-movable obstacles). However, the path may collide with movable obstacles.

3. `free_location`: this is done by randomly sampling a position such that there is a collision-free path from the obstacle's current position to the sampled position, without considering the reachability of the robot. It is implemented by an A* search assuming the object itself can move freely.

4. `close_to_object`: this is done by randomly sampling a position that is within the reach of the robot but also being close to the target object.

5. `collision_free_connect`: this is implemented again, by using A* search on the grid for the robot moving from one location to another. It considers collisions with both movable and non-movable obstacles.

6. `collision_free_connect_with_object`: this function generates a trajectory of the object assuming that the object `o` has been attached to the object (so they will move jointly).

There are two features of these samplers that are worth noting. First, these generators all depend on the state of the world (i.e., the position of the agent and the position of all other objects). Second, we do not explicitly assert the existence of certain values. For example, when we are trying to move an obstacle to a new position, we do not assume that there is a path connecting the current robot position and the obstacle position. Such "assertions" happen implicitly during the CSP-solving process. Essentially, if we can not find a valid assignment to a variable (a pose `x` that is close to the object), the plan trace refinement fails.

Since all these generators execute non-deterministically, for a fair comparison between all different methods, we set the maximum number of intermediate waypoints that can be selected by all algorithms to 1.

M.5 Manual of CDL

M.5.1 Key Concepts

Before we delve into the details of the programming language, we would like to start with a few key concepts in our design choices.

Object-centricness. In our language CDL, the state and the behavior representations are *lifted*. That is, we consider an environment composed of a set of entities. The state will be represented as features associated with entities or tuples of entities. Similarly, the behavior rules will optionally take entities as their parameters and achieve certain entity properties. This would allow us to define a state and behavior representation that is *compact* and can encode a *family* of decision-making problems of indefinite numbers of objects, which is crucial for decision-making in physical environments.

As an example, in CDL, a state feature `position_2d` can be defined as

```
typedef Object
feature position_1d(x: Object) -> float32
```

In this case, the feature is unary, representing a single floating-point number associated with each entity in the environment. A behavior rule `move_to` can be defined as:

```
behavior move_to(x: Object, target: float32):
body:
  do control_move(x, target)
eff:
  position_1d[x] = target
```

The effect of the behavior rule is defined as updating the `position_1d` property of the entity `x`.

Data-oblivious computation graphs. The second important concept in CDL is *data oblivious computation*. In a nutshell, a *data-obvilious* program is a program whose:

1. data accesses do not depend on the input values;
2. all functions that combine data values are encapsulated into black-box operations;
3. furthermore, the control flow depends only on the input size (optionally random values to enable stochastic computation).

A signature example of non-oblivious computation is recursion. If our termination condition depends on the value of the input and does not have a simple upper bound of

recursion depths, then we can not a priori construct the computation graph and apply it to any inputs. Actually, the decidability of the planning problem will also be undecidable (recall the halting problem, or Gödel's incompleteness theorem). Data-oblivious computation supports if-else conditions (which can be implemented by constructing both the `if` and the `else` branches), for-loops with constant bounds, or for-loops whose bounds only depends on the input size (the number of objects in the environment). It is a bit complex in construction but looping over all entities based on the value of a feature (smallest to biggest) can be implemented as a data-oblivious computation. But the compilation is inefficient, where the naïve implementation requires $O(N^2)$ circuit size. In practice, one may want to use other language features to realize this types of computation.

Data obliviousness will give us advantages in reasoning about policies or transition models written in the programming language. For example, if we consider the policy as a program that generates a sequence of primitive controller commands, representing it as a data-oblivious program would allow us to derive the sequence of behavior commands without binding the parameters. For example, the following program is data-oblivious:

```
behavior move_five_steps(x: Object): # oblivious
  body:
    for i in range(5):
      do move_to(x, pose[x] -1)
```

In essence, this program outputs a fixed sequence of 5 `move_to` primitive controller calls. By contrast, the following program is not data-oblivious:

```
behavior move_until_less_than_0(x: Object): # non-oblivious
  body:
    while pose[x] > 0: # CDL does not support while
      do move_to(x, pose[x] -1)
```

Because we can't decide how many primitive controllers will be generated by this program without knowing the actual value of the initial `pose[x]`. Furthermore, from a program verification perspective, data-oblivious programs enable a simple way to reason about the precondition and the postcondition of a program, which we will explore in detail in later sections.

M.5.2 State Representation: Types and Features

For each decision-making problem, CDL assumes that there is a fixed and finite set of entities. This is not a very strong limitation. For entity creation, one can create additional "hypothetical" entities in the initial state whose properties will be filled after the entity is

instantiated (although this assumes that we know an upper bound for the number of objects to be created). Similarly, for entity deletion, one can introduce a new unary feature `deleted` to record if an object has been deleted from the current state. Therefore, a world state, for any given problem, can be completely described in terms of values of a fixed set of basic *features* (of arity $0, 1, 2, \ldots$) of the given entities; the feature values may be continuous or discrete and may be high-dimensional.

In CDL, we make a distinction between *objects* (referring to entities in the physical world) and *values* (which are features or the return values of functions over these features). Both of them can be associated with types. An object type can be defined as:

```
typedef T: BaseT
```

where `T` is the type name and `BaseT` is the name of the base type it inherits. When `BaseT` is omitted, it refines to the basic entity type in CDL. This can also be explicitly declared by inheriting the builtin type `object`. For example:

```
typedef box
typedef robot
```

These statements are equivalent to:

```
typedef box: object
typedef robot: object
```

And, a value type can be defined as:

```
typedef pose_1d: float32
typedef pose_2d: vector[float32; 2]
typedef pose_nd: vector[float32]
typedef python_class1: pyobject
```

Here, we are demonstrating a few different primitive types that a value type can inherit from, including a vector type of a fixed dimension, a vector type of an unknown dimension (whose dimension will be determined during program execution), and value types that correspond to Python types in the host program. As we will see in the next section, the key difference between objects and values lies in whether we can *quantify* over them. For example, we can say, for all objects x of a certain object type in the state, $f(x)$ is true. However, we can not state that, for all possible values v of a value type, $f(v)$ is true, because the set of all possible v's will be an infinite set in general.

Based on types, we have the feature definition:

```
feature f(v: T, ...)-> RT,
```

where `f` is the name of the feature, `v: T` declares an argument of name `v` and of type `T`. All features have a finite number of arguments. `RT` is the value type of the feature.

M.5.3 Functions

There are two types of functions in CDL: primitive functions and "derived" functions. Primitive functions are functions without a body. Their actual implementation resides in the external program (e.g., Python):

```
def f(v: T, ...)-> RT
```

This definition defines a function of name `f` and have an argument `v` of type `T`. It returns a value of type `RT`.

On the contrary, derived functions have a body and can compose the results of features and the return value of other functions (primitive or derived).

```
def f(v: T, ...)-> RT:
  return ...
```

The function bodies must be data-oblivious. In particular, we support the following features:

1. Read of features.

```
feature_1[arg_1, arg_2] # or
feature_1(arg_1, arg_2)
```

2. Logic operators, including finite-universe quantification operations.

```
f(arg_1) and f(arg_2)
f(arg_1) or f(arg_2)
not f(arg_1)
f(arg_1) ^ f(arg_2) # XOR
forall o: T: f(o) # returns True if all entities of type T satisfies f
exists o: T: f(o) # returns True if at least one entity
                  # of type T satisfies f
```

3. Basic arithmetic operations such as +, -, *, /, ** (power).

```
f(arg_1) +f(arg_2)
f(arg_1) -f(arg_2)
f(arg_1) * f(arg_2)
f(arg_1) / f(arg_2)
f(arg_1) ** f(arg_2)
```

4. Assignments to local variables (`let`).

5. If-else conditions.

```
if c(...):
  statement_1
else:
  statement_2
```

6. "Parallel" for-loops over all entities that apply the same function over all entities. Assignments to local variables outside the scope of the for-loop is not allowed (`foreach`).

To ensure data-obliviousness, no self-recursion is supported. That is, it is not allowed to call a function `f` at any point of the execution of `f`. For example, the following definition of functions are invalid.

```
def f(x: float32) -> float32:
  return g(x)

def g(x: float32) -> float32:
  return f(x) # not allowed!
```

In this definition, even if we are not explicitly calling `f` itself inside `f`, we are implicitly creating self-recursion by calling `f` in a subroutine `g` that `f` invokes.

M.5.4 Non-Data-Oblivious Functions and Generators

In CDL, data-oblivious and non-oblivious functions can be mixed. Any data-oblivious computation can be illustrated as such a forward diagram, and in CDL, such diagrams will always have a finite depth (but maybe "wide" at each depth due to quantifiers). CDL does not impose any constraints on the primitive functions (f, g, and h) in this case, therefore they can be generally non-data-oblivious and they will be implemented in external Python scripts. Another important reason to use such external primitive functions is that they allow us to write sophisticated programs to reason about numeric features and parameters, which are crucial for physical decision-making problems.

If our goal is to have the feature c be a particular value (e.g., at a particular location), we need to find "good" values for features a, b, c and d at the previous timestep. In CDL, since the computation graph is independent of the feature values, this problem can be modeled as a general constraint satisfaction problem.

If we only have a "forward" model of f, g, and h, there will be little we can do with this optimization process. A general technique to tackle such optimization problems would be

to define additional inversion functions. Such functions are called "generators" in CDL. An example definition of a generator function is the following:

```
def f(x: float32, y: float32) -> float32

generator invert_f(x, y, z):
  goal: f(x, y) == z
  in: z
  out: x, y
```

It declares an external function (e.g., `invert_f`) which takes as input the value of `z` and output two values `x` and `y` and it is guaranteed that `f(x, y) == z`.

M.5.5 Behavior Effects

Combining all these features, an effect section can use all operations supported in CDL, call to primitive and derived functions, to assign values to features. Since we have constrained the programs to be data-oblivious, an advantage (in addition to being able to formulate the constraint satisfaction problem) is that we can now translate such imperative transition models into a descriptive model, to possibly support other types of reasoning.

To be precise, there are two conventional paradigms for modeling T_a, the transition function associated with behavior a: imperative models (e.g., Python programs) and descriptive models (e.g., PDDL programs). In an imperative model, the effect of a is modeled as a function $T_a(s)$ that takes in the current state and generates a new state s'.

On the contrary, in a descriptive model, we write down a Hoare logic sentence with preconditions and effects. It has two formulas pre_a and eff_a and asserts:

$$\forall s. \{pre_a(s)\} \implies \{\forall (s' \sim T_a(s)) . eff_a(s')\}.$$

That is, both the pre_a and eff_a are descriptors of the current state and the next state. From a perspective of formal verification, the precondition and effects characterize the execution of the program T_a, and therefore transform the planning problem into a mathematical proving problem.

Both approaches have their pros and cons. The straightforward advantage of the imperative approach is that writing T_a by hand is usually very easy. In particular, if we allow the forward program T_a to be as flexible as Turing-computable functions, then writing the descriptive model of T_a by hand can be very challenging: intuitively, this would require the program writer to reason carefully about all effects related to the generation of a sound plan, very much like defining a semantic type system. The important advantage of the Hoare logic model is that now the solver can go both forward and backward. In particular, if our goal is

to achieve a certain goal g, we can check the "effect" model of all actions to find relevant actions that can potentially achieve this goal.

The first empirical observation that we have about practical transition models (especially in physical decision-making problems) is that usually, we can constrain the forward model written in the imperative language (sacrificing the Turing completeness) but make it possible to derive Hoare logic statements from the forward model. The practice that CDL takes is that it restricts all effect sections to be data-oblivious. Furthermore, the corresponding circuit will always have a finite depth. CDL comes with a built-in compilation tool that can transform any function body into a holistic Haore logic description of their effects, which, subsequently, can be translated into a corresponding PDDL definition. For example, the following CDL program:

```
behavior do_something(x):
eff:
  for i in range(3):
    if f[x] > 0:
      f[x] -= 1
    else:
      break
```

can be translated into a precondition-effect modeling (in a PDDL-style language):

```
(:action
 :parameters (?x)
 :effect (and
   (= (f ?x)
     (condition (> (f ?x) 0)
       (condition (> (- (f ?x) 1) 0)
         (condition (> (- (- (f ?x) 1) 1) 0)
           (- (- (- (f ?x) 1) 1) 1)
           (- (- (f ?x) 1) 1)
         )
         (- (f ?x) 1)
       )
       (f ?x)
     )
   )
 )
```

```
)
```

M.5.6 Examples

The example below illustrates the ability to express a fine-grained ordering prior on subgoal achievement.

```
body:
  unordered:
    achieve open(cupboard)
    achieve holding(big_obj)
  let big_o_loc = loc_in(big_obj, cupboard)
  achieve loc[big_obj] == big_o_loc
  for each c: Cup:
    achieve contained_in(c, cupboard)
  achieve closed(cupboard)
```

The example below illustrates a CDL program for handling the classical STRIPS planning task of blocks world.

```
typedef block: object

# Features without return type annotations are assumed to be returning boolean values.

feature clear(x: block)
feature on(x: block, y: block)
feature on_table(x: block)
feature holding(x:block)
feature handempty()

controller pickup_table(x: block)
controller place_table(x: block)
controller stack(x: block, y: block)
controller unstack(x: block, y: block)

action r_holding_from_table(x: block):
  goal: holding(x)
  body:
    assert on_table(x)
    promotable sequential:
      achieve on_table(x)
```

```
      achieve clear(x)
    achieve handempty()
    pickup_table(x)
  eff:
    clear[x] = False
    handempty[...] = False
    on_table[x] = False
    holding[x] = True

action r_holding_from_stack(x: block):
  goal: holding(x)
  body:
    bind y: block where:
      on(x, y)
    promotable sequential:
      achieve on(x, y)
      achieve clear(x)
    achieve handempty()
    unstack(x, y)
  eff:
    clear[x] = False
    handempty[...] = False
    on[x, y] = False
    holding[y] = True
    clear[y] = True

action r_clear(x: block):
  goal: clear(x)
  body:
    bind y: block where:
      on(y, x)
    promotable sequential:
      achieve on(y, x)
      achieve clear(y)
    achieve handempty()
    unstack(y, x)
  eff:
    clear[y] = False
    handempty[...] = False
    on[y, x] = False
```

```
    holding[y] = True
    clear[x] = True

action r_clear_from_holding(x: block):
  goal: clear(x)
  body:
    assert holding(x)
    place_table(x)
  eff:
    holding[x] = False
    clear[x] = True
    handempty[...] = True
    on_table[x] = True

action r_handempty():
  goal: handempty()
  body:
    bind x: block where:
      holding(x)
    place_table(x)
  eff:
    holding[x] = False
    handempty[...] = True
    clear[x] = True
    on_table[x] = True

action r_on(x: block, y: block):
  goal: on(x, y)
  body:
    promotable sequential:
      achieve clear(y)
      achieve holding(x)
    stack(x, y)
  eff:
    clear[y] = False
    holding[x] = False
    on[x, y] = True
    clear[x] = True
    handempty[...] = True

```

```
action r_on_table(x: block):
  goal: on_table(x)
  body:
    promotable sequential:
      achieve holding(x)
    place_table(x)
  eff:
    holding[x] = False
    on_table[x] = True
    clear[x] = True
    handempty[...] = True
```

This definition would allow us to generate primitive plan solutions to planning problems such as the Sussman anomaly. Fig. M-1 shows the refinement process from the high-level goal into a sound plan trace. If we need to describe the behavior using only sequential programs (but not partial orders), it would require a dedicated rule for the tuple of three objects. This figure also illustrates the refinement operator $\mathcal{R}^*$ in action. We will see in the next section that this will allow us to construct a full spectrum between purely imperative behavior programming and purely declarative planning-based behavior generation.

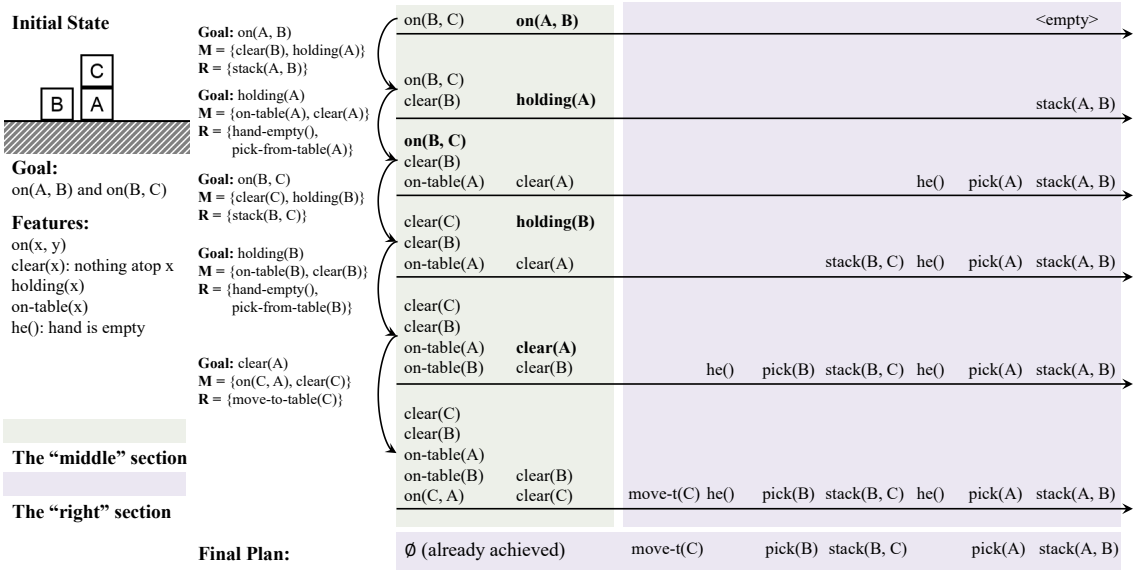


Figure M-1: **Example planning trace of CROW on the Sussman anomaly problem.** Leveraging the promotable "middle" section of programs, we can plan for primitive plans where some of the preconditions have to be achieved in parallel with other preconditions or subgoals. The figure shows a plausible refinement trace for a primitive plan for the Sussman anomaly [Sussman, 1973]. The refinement process starts with two subgoals $on(\mathrm{B}, \mathrm{C})$ and $on(\mathrm{A}, \mathrm{B})$, in this order. In the first step, it refines the subgoal $on(\mathrm{A}, \mathrm{B})$ using the associated behavior rule. This gives us a new $M = \{on(\mathrm{A}, \mathrm{B})), \{clear(\mathrm{B}), holding(\mathrm{A})\}_{seq}\}_{unordered}$ and a new $R = \{stack(\mathrm{A}, \mathrm{B})\}_{seq}$. In the second step, it nondeterministically selected $holding(\mathrm{A})$ for refinement, and produces a new $M' = \{on(\mathrm{A}, \mathrm{B})), clear(\mathrm{B}), \{on\text{-}table(\mathrm{A}), clear(\mathrm{A})\}_{seq}\}_{unordered}$ and a new $R' = \{hand\text{-}empty(), pick\text{-}from\text{-}table(\mathrm{A}), stack(\mathrm{A}, \mathrm{B})\}_{seq}$. This refinement process continues until we have a fully refined and sound plan.