

Program-Guided Framework for Interpreting and Acquiring Complex Skills with Learning Robots

by

Shao-Hua Sun

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Science)

March 2022

Table of Contents

List of Tables	ix
List of Figures	xi
Abstract	xv
I Introduction	1
Chapter 1: Introduction	2
1.1 Overview	2
1.1.1 Program Inference	4
1.1.1.1 Learning to Synthesize Programs from Demonstrations	4
1.1.1.2 Learning to Synthesize Programs from Reward Functions	5
1.1.2 Primitive Skill Acquisition	5
1.1.2.1 Meta-Learning & Meta-Reinforcement Learning	6
1.1.2.2 Learning from Demonstrations	6
1.1.3 Task Execution	7
1.1.3.1 Learning to Execute Programs	7
1.1.3.2 Learning to Compose Skills	7
1.2 Published Works	8
II Program Inference	11
Chapter 2: Learning to Synthesize Programs from Demonstrations	12
2.1 Introduction	12
2.2 Related Work	14
2.3 Problem Overview	15
2.4 Approach	17
2.4.1 Model Architecture	18
2.4.1.1 Demonstration Encoder	18
2.4.1.2 Summarizer Module	18
2.4.1.3 Program Decoder	20
2.4.2 Learning	20
2.4.3 Multi-task Objective	21
2.5 Experiments	22

2.5.1	Evaluation Metric	23
2.5.2	Evaluation Setting	24
2.5.3	Baselines	24
2.5.4	Karel	25
2.5.4.1	Environment and Dataset	25
2.5.4.2	Performance Evaluation	26
2.5.4.3	Effect of Summarizer	27
2.5.5	ViZDoom	27
2.5.5.1	Environment and Dataset	28
2.5.5.2	Performance Evaluation	28
2.5.5.3	Analysis	29
2.5.5.4	Debugging the Synthesized Program	30
2.6	Conclusion	30
2.7	Appendix	31
2.7.1	Detailed Network Architectures	31
2.7.1.1	Demonstration Encoder	31
2.7.1.2	Summarizer Module	31
2.7.1.3	Program Decoder	31
2.7.2	Training Details	32
2.7.3	One-shot Imitation Learning Baseline	32
2.7.4	Dataset Details	33
2.7.4.1	Karel	33
2.7.4.2	ViZDoom	34
Chapter 3: Learning to Synthesize Programs from Reward Functions		35
3.1	Introduction	35
3.2	Related Work	37
3.3	Problem Formulation	38
3.4	Approach	39
3.4.1	Learning a Program Embedding Space	40
3.4.1.1	Program Reconstruction	41
3.4.1.2	Program Behavior Reconstruction	41
3.4.1.3	Latent Behavior Reconstruction	42
3.4.2	Latent Program Search: Synthesizing a Task-Solving Program	43
3.5	Experiments	44
3.5.1	Karel Domain	44
3.5.2	Programs	45
3.5.3	Ablation Study	45
3.5.4	Baselines	48
3.5.5	Results	49
3.5.6	Generalization	51
3.5.7	Interpretability	52
3.6	Discussion	53
3.7	Appendix	54
3.7.1	Program Embedding Space Visualizations	54
3.7.2	Cross Entropy Method Trajectory Visualization	57
3.7.3	Program Embedding Space Interpolations	59
3.7.4	Program Evolution	61

3.7.5	Interpretability: Human Debugging of LEAPS Programs	61
3.7.6	Optimal and Synthesized Programs	64
3.7.6.1	Program Behavior Reconstruction	65
3.7.6.2	Karel Environment Tasks	65
3.7.7	Additional Generalization Experiments	66
3.7.7.1	Generalization on FOURCORNER, TOPOFF, and HARVESTER	66
3.7.7.2	Generalization to Unseen Configurations	67
3.7.8	Additional Analysis on Experimental Results	69
3.7.8.1	DRL vs. DRL-abs	69
3.7.8.2	VIPER Generalization	70
3.7.9	Detailed Descriptions and Illustrations of Ablations and Baselines	70
3.7.9.1	Ablations	71
3.7.9.2	Baselines	72
3.7.10	Program Dataset Generation Details	74
3.7.11	Karel Task Details	75
3.7.11.1	STAIRCLIMBER	75
3.7.11.2	FOURCORNER	77
3.7.11.3	TOPOFF	77
3.7.11.4	MAZE	77
3.7.11.5	CLEANHOUSE	77
3.7.11.6	HARVESTER	78
3.7.12	Hyperparameters and Training Details	78
3.7.12.1	DRL and DRL-abs	78
3.7.12.2	DRL-abs-t	80
3.7.12.3	HRL	81
3.7.12.4	Naïve	83
3.7.12.5	VIPER	85
3.7.12.6	Program Embedding Space VAE Model	86
3.7.12.7	Cross-Entropy Method (CEM)	89
3.7.12.8	Random Search LEAPS Ablation	91
3.7.13	Computational Resources	93
3.7.14	Toward Robotics Applications	94

III Primitive Skill Acquisition 112

Chapter 4: Meta-Learning on Multimodal Task Distributions	113	
4.1	Introduction	113
4.2	Related Work	115
4.3	Preliminaries	117
4.4	Method	118
4.4.1	Modulation Network	119
4.4.2	Task Network	120
4.5	Experiments	120
4.5.1	Regression Experiments	121
4.5.2	Image Classification	123
4.5.3	Reinforcement Learning	125
4.6	Conclusion	128

4.7	Appendix	129
4.7.1	Details on Modulation Operators	129
4.7.2	Further Discussion on Related Works	130
4.7.3	Baselines	130
4.7.4	Additional Experimental Details	131
4.7.4.1	Regression	131
4.7.4.2	Image Classification	133
4.7.4.3	Reinforcement Learning	138
4.7.5	Additional Experimental Results	140
4.7.5.1	Regression	140
4.7.5.2	Image Classification	142
4.7.5.3	Reinforcement Learning	142
Chapter 5:	Meta-Learning on Long-Horizon and Sparse-Reward Tasks	146
5.1	Introduction	146
5.2	Related Work	148
5.3	Problem Formulation and Preliminaries	149
5.4	Approach	151
5.4.1	Skill Extraction	152
5.4.2	Skill-based Meta-Training	152
5.4.3	Target Task Learning	154
5.5	Experiments	155
5.5.1	Experimental Setup	155
5.5.1.1	Maze Navigation	156
5.5.1.2	Kitchen Manipulation	157
5.5.2	Baselines	157
5.5.3	Results	158
5.5.4	Meta-Training Task Distribution Analysis	160
5.6	Conclusion	162
5.7	Appendix	162
5.7.1	Meta-Reinforcement Learning Method Ablation	162
5.7.2	Learning Efficiency on Target Tasks with Few Episodes of Experience	164
5.7.3	Investigating Offline Data vs. Target Domain Shift	165
5.7.4	Implementation Details on Our Method	168
5.7.4.1	Model Architecture	168
5.7.4.2	Training Details	169
5.7.5	Implementation Details on Baselines	170
5.7.5.1	SAC	170
5.7.5.2	PEARL and PEARL-ft	170
5.7.5.3	SPiRL	171
5.7.5.4	Multi-task RL (MTRL)	171
5.7.6	Meta-Training Tasks and Target Tasks	172
5.7.6.1	Maze Navigation	172
5.7.6.2	Kitchen Manipulation	172

Chapter 6: Learning from Observation	175
6.1 Introduction	175
6.2 Related Work	177
6.3 Method	179
6.3.1 Preliminaries	179
6.3.2 Learning Goal Proximity Function	180
6.3.3 Training Policy with Proximity Reward	181
6.4 Experiments	183
6.4.1 Experimental Setup	183
6.4.2 Baselines	184
6.4.3 Navigation	185
6.4.4 Maze2D	187
6.4.5 Ant Locomotion	188
6.4.6 Robotic Manipulation	188
6.4.7 Dexterous Hand Manipulation	189
6.4.8 Ablation Study	190
6.5 Conclusion	193
6.6 Appendix	194
6.6.1 Comparison with GAIL and Its Variants	194
6.6.2 Failure of GAIfO and SQIL	195
6.6.3 Analysis on Generalization of Our Method and Baselines	195
6.6.4 Further Ablations	198
6.6.5 Qualitative Results	200
6.6.6 Implementation Details	201
6.6.6.1 Environment Details	202
6.6.6.2 Network Architectures	203
6.6.6.3 Training Details	204

IV Task Execution 208

Chapter 7: Learning to Execute Programs	209
7.1 Introduction	209
7.2 Related Work	211
7.3 Problem Formulation	213
7.4 Approach	214
7.4.1 Program Interpreter	215
7.4.2 Perception Module	216
7.4.3 Policy	216
7.4.4 Learning	217
7.4.4.1 Perception Module	218
7.4.4.2 Policy	218
7.5 Experiments	219
7.5.1 Experimental Setups	219
7.5.1.1 Environment	219
7.5.1.2 Task Instructions	219
7.5.2 Training	220
7.5.3 End-to-end Learning Models	220

7.5.4	Results	221
7.5.4.1	Task Completion	221
7.5.4.2	Analysis	222
7.5.5	Policy Modulation	223
7.6	Conclusion	223
7.7	Appendix	224
7.7.1	Program Execution	224
7.7.2	DSL Design Principle	225
7.7.3	Extended Related Work	226
7.7.4	Discussions on Learned Modulation Mechanisms	227
7.7.5	Additional Experimental Details	228
7.7.5.1	Environment Details	228
7.7.5.2	Ground Truth Perceptions for End-to-end Learning Baselines	230
7.7.5.3	Task Instructions Details	231
7.7.5.4	Network Architectures	232
7.7.5.5	Raw RGB Input	234
7.7.5.6	Failure Analysis	235
7.7.5.7	Hyperparameters	239
7.7.5.8	Computational Resources	239
Chapter 8: Learning to Compose Skills		242
8.1	Introduction	242
8.2	Related Work	243
8.3	Approach	245
8.3.1	Preliminaries	246
8.3.2	Modular Framework with Transition Policies	246
8.3.3	Training Transition Policies	247
8.4	Experiments	250
8.4.1	Baselines	250
8.4.2	Robotic Manipulation	252
8.4.3	Locomotion	254
8.4.4	Ablation Study	256
8.4.5	Training of Transition Policy and Proximity Predictor	256
8.4.6	Visualizing Transition Trajectory	257
8.5	Conclusion	259
8.6	Appendix	260
8.6.1	Acquiring Primitive Policies	260
8.6.2	Training Details	260
8.6.2.1	Implementation Details	260
8.6.2.2	Replay Buffers	261
8.6.2.3	Proximity Reward	261
8.6.2.4	Proximity Predictor	262
8.6.2.5	Transition Policies	263
8.6.2.6	Scalability	265
8.6.3	Environment Descriptions	265
8.6.3.1	Robotic Manipulation	265
8.6.3.2	Locomotion	267

V Conclusion	271
Chapter 9: Conclusion	272
9.1 Summary	272
9.2 Future Directions	272
9.2.1 Program Inference	273
9.2.2 Primitive Skill Acquisition	274
9.2.3 Task Execution	274
Bibliography	275

List of Tables

2.1	Karel Results	25
2.2	Ablation Study on Summarizer Module	26
2.3	ViZDoom Results	29
2.4	ViZDoom If-else Results	29
3.1	Ablation Study	46
3.2	Program Embedding Space Smoothness	47
3.3	Results	49
3.4	Generalization Results	51
3.5	LEAPS Close Latent Program Interpolation	59
3.6	LEAPS Far Latent Program Interpolation	60
3.7	Program Evolution Over CEM Search	62
3.8	Human Debugging Experiment Results	63
3.9	Additional Generalization Results	66
3.10	Unseen Configurations Performance	68
3.11	Program Token Generation Probabilities	75
3.12	LEAPS Length 100 Synthesized Karel Programs	76
4.1	Regression Results	121
4.2	Classification Results	124

4.3	RL Results	128
4.4	Additional Regression Results	133
4.5	Classification Dataset Details	134
4.6	Hyperparameters for Classification	135
4.7	2-mode Classification Results	136
4.8	3-mode Classification Results	136
4.9	5-mode Classification Results	137
4.10	Hyperparameters for RL	140
6.1	Environment Details	205
6.2	Hyperparameters for Baselines	205
6.3	Hyperparameters for Our Method	207
7.1	Results	221
7.2	End-to-end Architecture Details	234
8.1	Robotic Manipulation Results	253
8.2	Locomotion Results	255
8.3	Hyperparameters	260

List of Figures

1.1	Overview of Program-Guided Framework for Interpreting and Acquiring Complex Skills with Learning Robots	2
2.1	Illustration of Neural Program Synthesis from Demonstration Videos	13
2.2	Domain-Specific Language	15
2.3	Model Overview	16
2.4	Summarizer Module	20
2.5	Karel Results	22
2.6	ViZDoom Results	27
2.7	Analysis on Varying Numbers of Demonstration Videos	30
3.1	Domain Specific Language	38
3.2	Framework Overview	40
3.3	Karel Problem Set	44
3.4	Visualizations of Learned Program Embedding Space	95
3.5	STAIRCLIMBER CEM Trajectory Visualization	96
3.6	FOURCORNER CEM Trajectory Visualization	97
3.7	Human Debugging Experiment User Interface	98
3.8	Human Debugging Experiment Example Programs (TopOFF)	99
3.9	Human Debugging Experiment Example Programs (FOURCORNER)	100
3.10	Human Debugging Experiment Example Programs (HARVESTER)	101

3.11	Ground-Truth Test Programs and Karel Programs	102
3.12	Program Reconstruction Task Synthesized Programs (naïve, LEAPS-P, LEAPS-P+R)	103
3.13	Program Reconstruction Task Synthesized Programs (LEAPS-P+L, LEAPS)	104
3.14	LEAPS Karel Tasks Synthesized Programs	105
3.15	LEAPS Ablations Illustrations	106
3.16	Baseline Methods Illustrations	107
3.17	Program Length Histograms	108
3.18	Karel Task Start/End State Depictions	109
3.19	Karel Rollout Visualizations	111
4.1	Model Overview	118
4.2	Regression Results	122
4.3	Visualization of Embedded Tasks	124
4.4	RL Environments	125
4.5	Point Mass Results	126
4.6	REACHER and ANT Results	126
4.7	Visualization of Embedded Regression Tasks	131
4.8	Visualization of Regression Results	133
4.9	Classification Dataset Summary	134
4.10	Visualization of Embedded Classification Tasks	137
4.11	Training Curves on RL Tasks	138
4.12	Additional Regression Results	141
4.13	Additional Point Mass Results	143
4.14	MAML and Multimodal MAML Training Curves on Classification Tasks	144
4.15	Multi-MAML Training Curves on Classification Tasks	145

5.1	Overview	146
5.2	Method Overview	151
5.3	Environments	156
5.4	Target Task Learning Efficiency	159
5.5	Qualitative Results	160
5.6	Meta-Training Task Distribution Analysis	161
5.7	Task Distributions for Task Length Ablation	164
5.8	Meta-Training Performance for Task Length Ablation	165
5.9	Qualitative Result of Meta-RL Method Ablation	166
5.10	Performance with Few Episodes of Target Task Interaction	167
5.11	Image-Based Maze Navigation with Distribution Shift	167
5.12	Maze Meta-Training and Target Task Distributions	172
5.13	Maze Meta-Training and Target Task Distributions for Meta-training Task Distribution Analysis	172
6.1	Framework Overview	176
6.2	Environments	183
6.3	Goal Completion Rate	185
6.4	Generalization Analysis	187
6.5	Goal Proximity Function Analysis	189
6.6	Ablation Study	190
6.7	Analysis on Generalizing to Unseen States	197
6.8	Additional Ablation Study	199
6.9	Analysis on Proximity Discounting Factor	200
6.10	Effect of Spectral Normalization	201
6.11	Proximity Prediction Visualization	206

6.12	NAVIGATION 25% heldout Set	207
7.1	Overview	211
7.2	Domain-Specific Language	211
7.3	Framework Overview	214
7.4	Learning a Multitask Policy via Learned Modulation	217
7.5	Analysis on End-to-end Learning Models	222
7.6	Environment Example	230
7.7	First Failure Rate of Subtasks	236
7.8	Average Time Cost of Subtasks	236
7.9	Additional Analysis on Completion Rates	237
7.10	Exemplar Data and Language Ambiguity	239
7.11	Training Program Statistics	240
7.12	Testing Program Statistics	240
7.13	Complex Testing Program Statistics	241
8.1	Transition Policy Illustration	244
8.2	Framework Overview	245
8.3	Training Overview	248
8.4	Results	251
8.5	Average Transition Length and Proximity Reward	257
8.6	Visualization of Transition Trajectories	258
8.7	Ablation Study on Proximity Discounting Factor	262

Abstract

Recent development in artificial intelligence and machine learning has remarkably advanced machines' ability to understand images and videos, comprehend natural languages and speech, and outperform human experts in complex games. However, building intelligent robots that can physically interact with their surroundings as well as learn to operate in unstructured environments, manipulate unknown objects, and acquire novel skills – to free humans from tedious or dangerous manual work – remains challenging. The focus of my research is to develop a robot learning framework that enables robots to acquire long-horizon and complex skills with hierarchical structures, such as furniture assembly and cooking. Specifically, I aim to devise a robot learning framework which is: (1) interpretable: by decoupling interpreting skill specifications (e.g. demonstrations, reward functions) and executing skills, (2) programmatic: by generalizing from simple instances to complex instances without additional learning, (3) hierarchical: by operating on a proper level of abstraction that enables human users to interpret high-level plans of robots allows for composing primitive skills to solve long-horizon tasks, and (4) modular: by being equipped with modules specialized in different functions (e.g. perception, action) which collaborate, allowing for better generalization. This dissertation discusses a series of research projects toward building such an interpretable, programmatic, hierarchical, and modular robot learning framework.

Part I

Introduction

Chapter 1

Introduction

1.1 Overview

Recent advancement in artificial intelligence and machine learning has remarkably advanced machines' ability to understand images and videos (e.g. object detection, semantic segmentation, action recognition, image captioning), comprehend natural languages and speech (e.g. machine translation, document summarization, speech recognition), and even outperform human experts in complex games (e.g. Go, Dota 2, StarCraft II). With the ability to learn, those systems generalize reasonably well on a wide range of tasks, and some have even been deployed as widely used products. However, building reliable artificial

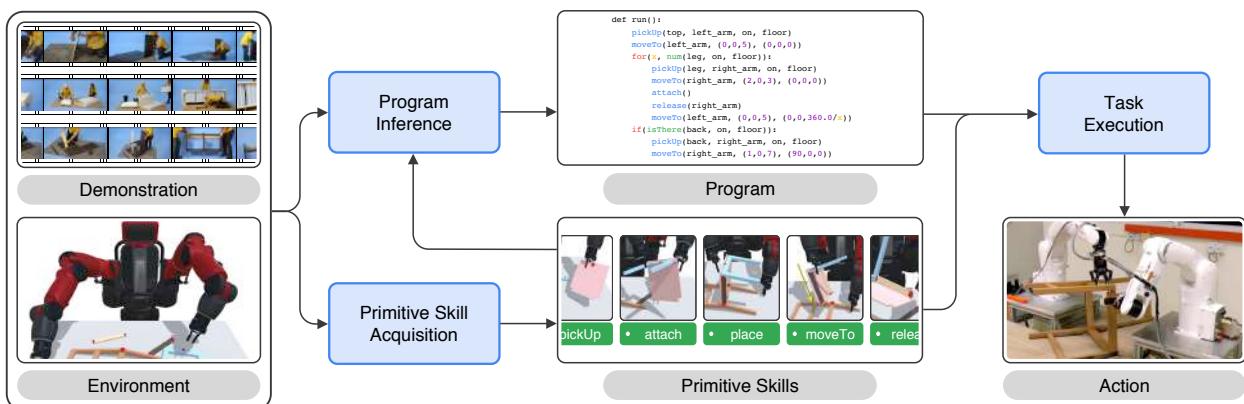


Figure 1.1: An overview of the proposed program-guided framework for interpreting and acquiring complex skills with learning robots. Learning and inference modules (in blue) connect demonstration, environment, program, primitive skills, and executable action (in black).

intelligence agents (*i.e.* robots) that can physically interact with their surroundings while learning to operate in unstructured environments, manipulate unseen objects, and acquire novel skills – to free humans from tedious or dangerous manual work – remains challenging.

Recently, the success of deep reinforcement learning (DRL) has led many researchers to develop learning frameworks to control robots. Compared to traditional robotics pipelines, DRL methods approximate policies using deep neural networks, which are optimized to automate the process of designing sensing, planning, and control algorithms by letting robots learn in an end-to-end fashion. Despite the recent progress in the field, such neural network policies suffer from several fundamental issues. First, such black-box policies are not interpretable to humans and therefore are difficult to debug when robots fail to perform a task. Second, acquiring complex skills through trial and error still remains challenging and these neural network policies often have difficulty generalizing to novel scenarios. Third, most existing works are limited to short-horizon skills such as pushing and picking up objects. Finally, most approaches are designed to acquire skills from scratch instead of building upon previously learned skills.

The focus of my research is to develop a robot learning framework that allows robots to acquire long-horizon and complex skills with hierarchical structures such as furniture assembly and cooking. Specifically, I aim to devise a robot learning framework which is: (1) **interpretable**: by decoupling interpreting skill specifications (*e.g.* demonstrations, reward functions) and executing skills, (2) **programmatic**: by generalizing from simple instances to complex instances without additional training, (3) **hierarchical**: by operating on a proper level of abstraction that enables human users to interpret high-level plans of robots allows for composing primitive skills to solve long-horizon tasks, and (4) **modular**: by being equipped with modules specialized in different functions (*e.g.* perception, action) which collaborate together, allowing for better generalization.

To this end, I present a robot learning framework which represents desired behaviors as a *program* as well as acquires and utilizes *primitive skills* for learning to execute desired skills, as shown in Figure 1.1.

Specifically, instead of learning in an end-to-end manner, I propose to design specialized learning modules that aim to (1) perform **program inference** to explicitly infer underlying programs that describe the skills of interest, (2) **acquire primitive skills** that can be used to compose more complex and longer-horizon skills, and (3) perform **task execution** by following the inferred program and utilizing acquired primitive skills to replicate the desired skills. In the following, I will discuss the details of each module.

1.1.1 Program Inference

I propose to utilize *programs*, structured in a formal domain-specific language, to describe behaviors (see an example program in Figure 1.1). Programs are not only interpretable to humans, but also more directly machine-executable since they are less ambiguous compared to natural languages. However, specifying a task or a desired behavior by writing a program requires substantial expertise and can be tedious. Therefore, I propose to learn to perform *program inference*, which aims to construct a program that describes a task-solving procedure from task specifications which are easier to provide (e.g. demonstrations, reward functions). In the following, I will describe the projects carried out by my collaborators and me, which focus on learning to perform program inference from such task specifications.

1.1.1.1 Learning to Synthesize Programs from Demonstrations

In [286], we aim to interpret the provided demonstrations and infer the intended skill by learning to explicitly synthesizing the underlying program which describes the skill in a structured language. A synthesized program explicitly describes diverse situations and the corresponding subtasks to execute. Moreover, a program describes tasks and their hierarchies at a set level of abstraction (e.g. actions such as `moveTo`, `attach` and perceptions such as `isThere`), providing scaffolding for learning long-horizon, hierarchically structured tasks. Specifically, in [286], we study the problem of mimicking behaviors presented in a set of demonstration videos, where demonstrator's behaviors vary from video to video due to different

environmental conditions. To address it, we propose to synthesize a program from demonstration videos and then execute the program to replicate the demonstrator’s behavior. The results suggest that learning to explicitly synthesize programs instead of learning a black-box policy encourages the model to pay extra attention to the decision-making logic of the demonstrator, leading to its superior performance.

1.1.1.2 Learning to Synthesize Programs from Reward Functions

While [286] achieves promising results on learning to synthesize programs from demonstrations, obtaining demonstrations can sometimes be expensive or even impossible. Therefore, in [303], we aim to devise a framework that can learn to perform program inference directly from reward signals provided by a Markov decision process (MDP). To alleviate the difficulty of learning to compose programs to induce the desired agent behavior from scratch, we propose to first learn a program embedding space that continuously parameterizes diverse behaviors in an unsupervised manner and then search over the learned program embedding space to yield a program that maximizes the return for a given task. The results suggest that the proposed framework can produce interpretable and more generalizable policies which outperform DRL methods.

Beside inferring programs from a variety of task specifications, our recent work explores synthesizing a program by hierarchically composing multiple programs. This allows for synthesizing programs that are long and deeply nested, which can induce more complex behaviors, and therefore increases the applicability of representing behaviors using programs.

1.1.2 Primitive Skill Acquisition

The aim of this stage is to robustly and efficiently acquire a set of primitive skills (*e.g.* `moveTo`, `attach`, `place` in Figure 1.1) that can be used to compose more complex skills to enable executing behaviors specified in programs obtained from the program inference stage. To this end, my research focuses on

meta-learning and learning from demonstrations. In the following, I will describe the projects carried out by my collaborators and me, which focus on these two directions.

1.1.2.1 Meta-Learning & Meta-Reinforcement Learning

Meta-Learning on Multimodal Task Distributions. To efficiently acquire a diverse set of skills, we propose to leverage the recent advancement in meta-learning and meta-reinforcement learning. Specifically, we aim to leverage model-agnostic meta-learning (MAML), which allows an agent to learn from a distribution of tasks and then quickly adapt to novel tasks with few gradient updates. Yet, MAML seeks a common initialization shared across the entire task distribution, substantially limiting the diversity of the task distributions that they can learn from. To enable an agent to adapt to a diverse set of primitive skills, we propose a multimodal MAML framework [314, 315], which can modulate its meta-learned prior parameters according to the identified task families, allowing more efficient fast adaptation. The proposed multimodal MAML framework achieves superior performance on not only reinforcement learning but also few-shot regression and image classification.

Meta-Learning on Long-Horizon and Sparse-Reward Tasks. In our recent works [205, 206, 207], we aim to address a common issue of most existing meta-reinforcement learning methods – they are limited to short-horizon tasks with dense rewards. To this end, we propose to first extract composable skills and a skill prior from agent play data in the form of offline datasets, which then enables meta-learning on long-horizon, sparse-reward tasks.

1.1.2.2 Learning from Demonstrations

Learning from Observation. Another research direction that aims to improve sample efficiency for acquiring skills is learning from demonstrations. However, demonstrators' actions might not always be available and how a learning agent can generalize beyond situations seen in demonstrations remains

challenging. In [161], we study learning from observation (*i.e.* imitate demonstrators without accessing to their actions but only state sequences) and generalization to novel situations beyond demonstrations. Specifically, we proposed to learn a task progress estimator and use the task progress estimate as a dense reward for training a policy. We show that our proposed method can robustly generalize compared to prior methods on a set of tasks in navigation, locomotion, and robotic manipulation – even with demonstrations that cover only part of the states.

1.1.3 Task Execution

To replicate a skill by following an inferred program (Section 1.1.1) and utilizing a set of acquired primitive skills (Section 1.1.2), an agent needs to (1) deduce the correct order of primitive skills that need to be executed as well as (2) smoothly chain them together.

1.1.3.1 Learning to Execute Programs

To address (1), we propose a framework that learns to interpret and follow a program by employing a perception module which learns to decide which path in a program should be taken and a policy which fulfills each desired primitive skill in [287].

1.1.3.2 Learning to Compose Skills

For (2), when the primitive skills were independently obtained and the environment dynamic is complex (*e.g.* continuous control), simply sequentially executing the primitive skills can often fail. Therefore, we propose to learn transition policies which effectively navigate from an ending state of any primitive skill to suitable initial states of the following primitive skill in [160].

1.2 Published Works

This dissertation presents a number of techniques for allowing learning robots to interpret and acquire complex skills, which were presented at top-tier computer science and machine learning venues. This section enumerates these published works, and briefly describes the content of each chapter in this dissertation.

Part II: Program Inference

- Chapter 2 corresponds to a paper [286] published at International Conference on Machine Learning (ICML) 2018. It aims to infer decision making logic in demonstration videos, allowing for accurately imitating demonstrator's behaviors. To this end, we propose a framework that is able to explicitly synthesize underlying programs, that describe demonstrator decision making logic, from behaviorally diverse and visually complicated demonstration videos.
- Chapter 3 corresponds to a paper [303] published at Neural Information Processing Systems (NeurIPS) 2021. It presents a framework that learns to synthesize a program, detailing the procedure to solve a task in a flexible and expressive manner, solely from reward signals. To alleviate the difficulty of learning to compose programs to induce the desired agent behavior from scratch, we propose to learn a program embedding space that continuously parameterizes diverse behaviors in an unsupervised manner and then search over the learned program embedding space to yield a program that maximizes the return for a given task.

Part III: Primitive Skill Acquisition

- Chapter 4 corresponds to a paper [314] published at Neural Information Processing Systems (NeurIPS) 2019 and a paper [315] presented in Meta-Learning Workshop at Neural Information Processing Systems (NeurIPS) 2018. Model-agnostic meta-learners aim to acquire meta-prior parameters from a distribution of tasks and adapt to novel tasks with few gradient updates. Yet, seeking a common initialization shared across the entire task distribution substantially limits the diversity of the task

distributions that they are able to learn from. We propose a multimodal MAML (MMAML) framework, which is able to modulate its meta-learned prior according to the identified mode, allowing more efficient fast adaptation.

- Chapter 5 corresponds to a paper [207] published at International Conference on Learning Representations (ICLR) 2022, a paper [205] presented in Meta-Learning Workshop at Neural Information Processing Systems (NeurIPS) 2021, and a paper [206] presented in Deep RL Workshop at Neural Information Processing Systems (NeurIPS) 2021. We devise a method that enables meta-learning on long-horizon, sparse-reward tasks, allowing us to solve unseen target tasks with orders of magnitude fewer environment interactions. Specifically, we propose to (1) extract reusable skills and a skill prior from offline datasets, (2) meta-train a high-level policy that learns to efficiently compose learned skills into long-horizon behaviors, and (3) rapidly adapt the meta-trained policy to solve an unseen target task.
- Chapter 6 corresponds to a paper [161] published at Neural Information Processing Systems (NeurIPS) 2021. Task progress is intuitive and readily available task information that can guide an agent closer to the desired goal. Furthermore, a progress estimator can generalize to new situations. From this intuition, we propose a simple yet effective imitation learning from observation method for a goal-directed task using a learned goal proximity function as a task progress estimator, for better generalization to unseen states and goals. We obtain this goal proximity function from expert demonstrations and online agent experience, and then use the learned goal proximity as a dense reward for policy training.

Part IV: Task Execution

- Chapter 7 corresponds to a paper [287] published at International Conference on Learning Representations (ICLR) 2020. We propose to utilize programs, structured in a formal language, as a precise

and expressive way to specify tasks, instead of natural languages which can often be ambiguous. We then devise a modular framework that learns to perform a task specified by a program – as different circumstances give rise to diverse ways to accomplish the task, our framework can perceive which circumstance it is currently under, and instruct a multitask policy accordingly to fulfill each subtask of the overall task.

- Chapter 8 corresponds to a paper [160] published at International Conference on Learning Representations (ICLR) 2019. Humans acquire complex skills by exploiting previously learned skills and making transitions between them. To empower machines with this ability, we propose a method that can learn transition policies which effectively connect primitive skills to perform sequential tasks without handcrafted rewards. To efficiently train our transition policies, we introduce proximity predictors which induce rewards gauging proximity to suitable initial states for the next skill.

Part II

Program Inference

Chapter 2

Learning to Synthesize Programs from Demonstrations

2.1 Introduction

Imagine you are watching others driving cars. You will easily notice many common behaviors even if you know nothing about driving. For example, cars stop when a traffic light turns to red and move again when the light turns to green. Cars also slow down when drivers see a pedestrian jay-walking. Just like in this example, humans can abstract behaviors – especially extracting the structural relationship between action and perception (e.g. light, pedestrian).

Can machines also reason decision making logic behind behaviors? While there has been tremendous effort and success in understanding videos, they have been mostly focused on recognizing actions, finding and naming objects, or predicting future outcomes. However, the problem of reasoning decision making logic behind behaviors is a crucial skill for machines to mimic and collaborate with humans. Hence, our goal is to step towards developing a method that can interpret perception-based decision making logic from visual behavior demonstrations.

Our insight is to exploit declarative programs, structured in a formal language, as behavior interpreting representations. The formal language is composed of action blocks, perception blocks, and control flow (e.g. if/else). Programs written in such a language can explicitly model the connection between an observation (e.g. traffic light, biker) and an action (e.g. stop). An example is shown in Figure 2.1. Described in a formal

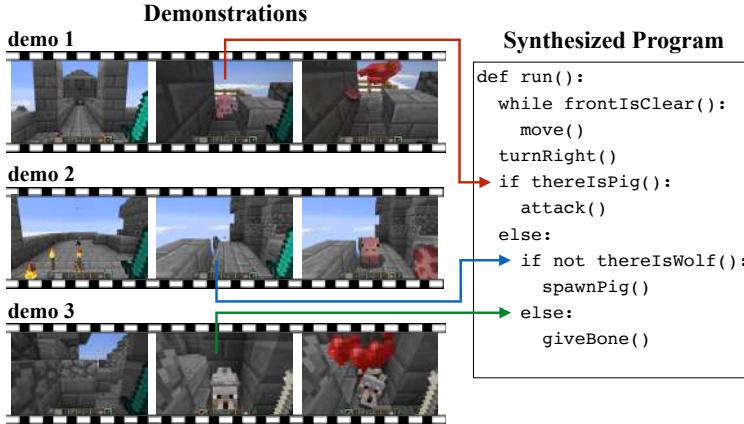


Figure 2.1: An illustration of neural program synthesis from demonstration videos. Given multiple demonstration videos exhibiting diverse behaviors, our neural program synthesizer learn to produce interpretable and executable underlying programs. Divergence above occurs based on perception in the second frame.

language, programs are logically interpretable and executable. Thus, the problem of interpreting decision making logic from visual demonstrations can be reduced to extracting an underlying program.

In fact, there have been many neural network frameworks proposed recently for program induction or synthesis. First, a variety of frameworks [132, 247, 331, 61] are proposed to induce latent representations of underlying programs. While they can be efficient at mimicking desired behaviors, they do not explicitly yield interpretable programs, resulting in inexplicable failure cases. On the other hand, there is another line of work directly synthesizing programs [63, 35] giving full interpretability. These approaches have shown highly competitive results in simple input/output pairs but are limited in the expressibility of programs. Hence, in this paper, we further extend their models with our model to synthesize programs while handling complex visual sequential inputs.

Our goal is to interpret logics behind various visual demonstrations. In other words, we want a model that can synthesize programs mimicking behaviors in demonstrations. Therefore, the model is required to handle diverse sequential visual data. To address this requirement, we propose a program synthesis architecture augmented with a summarizer module – a module that is capable of encoding the interrelation between multiple demonstrations and summarizes them into a compact vector representation. The

summarizer module empowers the model to handle a varying number of demonstrations, resulting in extra flexibility.

We extensively evaluate our model in two distinct environments: a fully observable, third-person environment (Karel) and a partially observable, egocentric game (ViZDoom). Our experiments in both environments show that directly modeling a behavior as a program has several benefits such as a better reasoning of the underlying conditions behind action compared against other methods. We also present an additional strength of our approach such as interpretability that enables human interaction for fixing and debugging.

In summary, in this paper, we introduce a novel problem of program synthesis from demonstrations of sequential visual data and a method to address it. This ultimately enables machines to explicitly interpret decision making logic and further interact with humans through a debugging process. We also demonstrate that our algorithm can synthesize programs reliably on multiple environments.

2.2 Related Work

Program Induction Learning to perform a specific task by inducing latent representations of underlying task-specific programs is known as *program induction*. Various approaches have been developed: designing end-to-end differentiable architectures [95, 96, 346, 132, 131, 97, 208], learning to call subprograms using step-by-step supervision [247, 38], and few-shot program induction [61]. Contrary to our work, those method do not return explicit programs.

Program Synthesis The line of work in *program synthesis* focuses on explicitly producing programs that are restricted to certain languages. [24] train a model to predict program attributes and used external search algorithms for inductive program synthesis. [219, 63] directly synthesize simple string transformation programs. [35] employ reinforcement learning to directly optimize the execution of generated programs.

```

Program  $m := \text{def run}() : s$ 
Statement  $s := \text{while}(b) : (s) \mid s_1; s_2 \mid a \mid \text{repeat}(r) : (s)$ 
 $\quad \mid \text{if}(b) : (s) \mid \text{ifelse}(b) : (s_1) \text{ else } (s_2)$ 
Repetition  $r := \text{Number of repetitions}$ 
Condition  $b := \text{percept} \mid \text{not } b$ 
Perception  $p := \text{Domain dependent perception primitives}$ 
Action  $a := \text{Domain dependent action primitives}$ 

```

Figure 2.2: Domain-specific language for the program representation. The program is composed of domain dependent perception and action primitives and control flows.

However, those methods are limited to synthesizing programs from input-output pairs, which substantially restricts the expressibility of the programs that are considered; instead, we address the problem of synthesizing programs from full demonstrations videos.

Imitation Learning The methods that are concerned with acquiring skills from expert demonstrations, dubbed *imitation learning*, can be split into *behavioral cloning* [236, 237, 251] which casts the problem as a supervised learning task and *inverse reinforcement learning* [210] that extracts estimated reward functions given demonstrations. Recently, [69, 82, 331] have studied the task of mimicking given few demonstrations. This line of work can be considered as *program induction*, as they imitate demonstrations without explicitly modeling underlying programs. While those methods are able to mimic given few demonstrations, it is not clear if they could deal with multiple demonstrations with diverse branching conditions.

2.3 Problem Overview

In this section, we define our formulation for program synthesis from diverse demonstration videos. We would like to comprehend and replicate demonstrated behaviors by revealing the formal structure between demonstrators' perception and actions. We therefore formally define programs in a domain-specific language (DSL) with perception primitives, action primitives and control flows. Action primitives define the way that

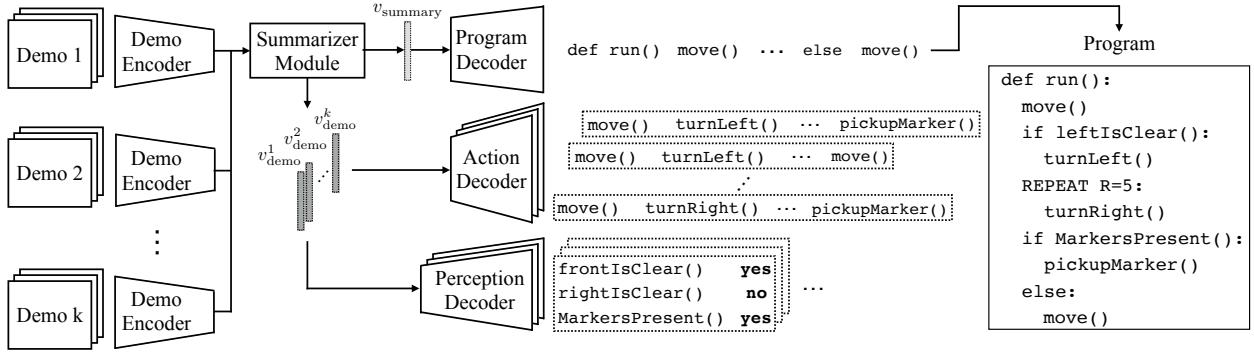


Figure 2.3: Model Architecture. The demonstration encoder encodes each of the k demonstrations separately and the summarizer network aggregates them to construct a summary vector. The summary vector is used by the program decoder to produce program tokens sequentially. The encoded demonstrations are used to decode the action sequence and perception conditions as additional supervision.

agents can interact with an environment, while perception primitives describe how agents can percept it.

On the other hand, control flows can include if/else statements, while loops, repeat statements, and simple logic operations. An example of control flows introduced in [226] is shown in Figure 2.2. Note that we focus on perceptions with boolean returns in this paper, while more generic perception type constraint is possible.

A program η is a deterministic function that outputs an action $a \in \mathcal{A}$ given a history of states at time step t , $H_t = (s_1, s_2, \dots, s_t)$, where $s \in \mathcal{S}$ is a state of the environment. The generation of an action given the history of states is represented as $a_t = \eta(H_t)$. In this paper, we focus on a program type that can be represented in DSL by a code $C = (w_1, w_2, \dots, w_N)$, which is a sequence of tokens w .

A demonstration $\tau = ((s_1, a_1), (s_2, a_2), \dots, (s_T, a_T))$ is a sequence of state and action tuples generated by an underlying program η^* given a initial state s_1 . Given an initial state s_1 and its corresponding state history H_1 , the program generates new action $a_1 = \eta^*(H_1)$. The following state s_2 is generated by a state transition function T : $s_2 \sim T(s_1, a_1)$. The newly sampled state is incorporated into the state history $H_2 = H_1 \cup (s_2)$ and this process is iterated until the end of file action $\text{EOF} \in \mathcal{A}$ is returned by the program. A set of demonstrations $D = \{\tau_1, \tau_2, \dots, \tau_K\}$ can be generated by running a single program η^*

on different initial states $s_1^1, s_1^2, \dots, s_1^K$, where each initial state is sampled from a initial state distribution (i.e. $s_1^K \sim P_0(s_1)$).

While we are interested in inferring a program η^* from a set of demonstrations D , it is preferable to predict a code C^* instead, because it is a more accessible representation while immediately convertible to a program. Formally, we formulate the problem as a sequence prediction where the input is a set of demonstrations D and the output is a code C . Note that our objective is not about inferring a code perfectly but instead generating a code that can infer a program. This requires a carefully chosen measure for successful code synthesis, discussed in Section 2.5.1.

2.4 Approach

Inferring a program behind a set of demonstrations requires (1) interpreting each demonstration video (2) spotting and summarizing the difference among demonstrations to infer the conditions behind the taken actions (3) describing the understanding of demonstrations in a written language, Based on this intuition, we design a neural architecture composed of three components:

- **Demonstration Encoder** receives a demonstration video as input and produces an embedding that captures an agent's actions and perception.
- **Summarizer Module** discovers and summarizes where actions diverge between demonstrations and upon which branching conditions subsequent actions are taken.
- **Program Decoder** represents the summarized understanding of demonstrations as a code sequence.

The details of the three main components are described in the Section 2.4.1, and the learning objective of the proposed model is described in Section 2.4.2. Section 2.4.3 introduces auxiliary tasks for encouraging the model to learn the knowledge that is essential to infer a program.

2.4.1 Model Architecture

Figure 2.3 illustrates the overall architecture of the proposed model, which is composed of demonstration encoders, a summarizer module, and a program decoder. Details of each component are described in the following sections.

2.4.1.1 Demonstration Encoder

The demonstration encoder performs two types of understanding over a single demonstration. The first is understanding visible actions in each time steps and the second is summarizing the overall action sequence in a demonstration as a single idea. The demonstration encoder performs both types of understanding at the same time using a state encoder and an LSTM (Long Short Term Memory) [115].

The state encoder, a stack of convolutional layers, encodes a state s_t to its embedding as a state vector $v_{\text{state}}^t = \text{CNN}_{\text{enc}}(s_t) \in \mathbb{R}^d$, where $t \in [1, T]$ is the time-step.

The LSTM encodes each state representation and summarized representation at the same time.

$$c_{\text{enc}}^t, h_{\text{enc}}^t = \text{LSTM}_{\text{enc}}(v_{\text{state}}^t, c_{\text{enc}}^{t-1}, h_{\text{enc}}^{t-1}), \quad (2.1)$$

where, $t \in [1, T]$ is the time step, c_{enc}^t is the cell state, h_{enc}^t is the hidden state. Both the final state tuples $(c_{\text{enc}}^T, h_{\text{enc}}^T)$ encode the overall idea of the demonstration and intermediate hidden states $\{h_{\text{enc}}^1, h_{\text{enc}}^2, \dots, h_{\text{enc}}^T\}$ containing high level understanding of each state and are used as an input to the following modules.

2.4.1.2 Summarizer Module

The summarizer module first reviews each demonstration again with the context of the whole demonstrations to infer underlying conditions behind visible actions. The inferred conditions are summarized within a demonstration and then summarized again over multiple demonstrations. An illustration of the summarizer is shown in Figure 2.4.

The first summarization is performed by a reviewer module, an LSTM initialized with the pooled final state tuples of the demonstration encoder. The pooled final state tuples of the demonstration encoder is formally written as follows

$$c_{\text{review}}^0 = \frac{1}{K} \sum_{k=1}^K c_{\text{enc}}^{T,k}, \quad h_{\text{review}}^0 = \frac{1}{K} \sum_{k=1}^K h_{\text{enc}}^{T,k}, \quad (2.2)$$

where $(c_{\text{enc}}^{T,k}, h_{\text{enc}}^{T,k})$ is the final state tuple of the k th demonstration encoder. Then the reviewer LSTM encodes the hidden states

$$c_{\text{review}}^t, h_{\text{review}}^t = \text{LSTM}_{\text{review}}(h_{\text{enc}}^t, c_{\text{review}}^{t-1}, h_{\text{review}}^{t-1}), \quad (2.3)$$

where the final hidden state becomes a demonstration vector $v_{\text{demo}} = h_{\text{review}}^T \in \mathbb{R}^d$, which includes the summarized information within a single demonstration.

The final summarization, which is performed over multiple demonstrations, is performed by an aggregation module. The aggregation module gets K demonstration vectors and summarize them into a single compact vector representation. To effectively model complex relations between different demonstrations, we employ relational networks (RN) [257]. The aggregation process is formally written as follows.

$$v_{\text{summary}} = \text{RN}(v_{\text{demo}}^1, v_{\text{demo}}^2, \dots, v_{\text{demo}}^K), \quad (2.4)$$

where $v_{\text{summary}} \in \mathbb{R}^d$ is the the summarized demonstration vector. We show that employing the summarizer module significantly alleviate the difficulty of handling multiple demonstrations and improve generalization over different number of generations in Section 2.5 .

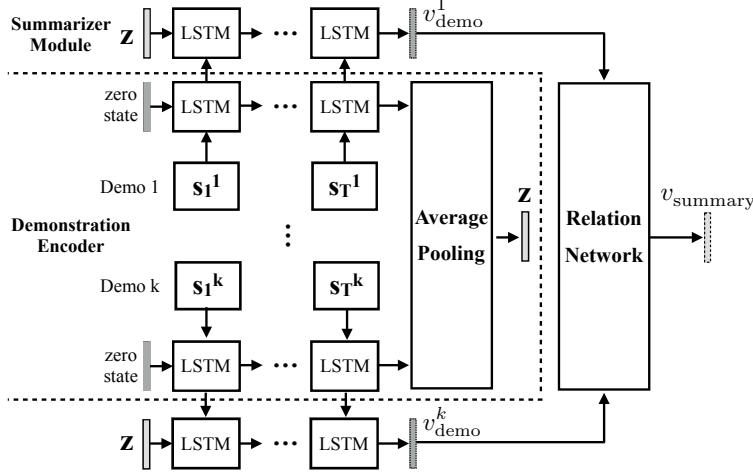


Figure 2.4: Summarizer Module. The demonstration encoder (inner layer) encodes each demonstration starting from a zero state. The summarizer module (outer layer) aggregates the outputs of the demonstration encoder with a relation network to provide context from other demonstrations.

2.4.1.3 Program Decoder

The program decoder generates programs from a summarized vector representation of the demonstrations.

We use LSTMs similar to [289, 311] as a program decoder. Initialized with the summary vector v_{summary} , the LSTM at each time step gets the previous token embedding as an input and outputs a probability of the following program tokens as in the Equation 2.5. During training, the previous ground truth token is fed as an input, and during inference, the predicted token in the previous steps is fed as an input.

2.4.2 Learning

The proposed model learns a conditional distribution between a set of demonstrations D and a corresponding code $C = \{w_1, w_2, \dots, w_N\}$. By employing the LSTM program decoder, this problem becomes an autoregressive sequence prediction [289]. For a given demonstration and previous code token w_{i-1} , our model is trained to predict the following ground truth token w_i^* , where the cross entropy loss is optimized.

$$\mathcal{L}_{\text{code}} = -\frac{1}{NM} \sum_{m=1}^M \sum_{n=1}^N \log p(w_{m,n}^* | W_{m,n-1}^m, D_m), \quad (2.5)$$

where M is the total number of training examples, $w_{m,n}$ is the n th token of the m th training example and D_m are m th training demonstrations. $W_{m,n} = \{w_{m,1}, \dots, w_{m,n}\}$ is the history of previous token inputs at time step n .

2.4.3 Multi-task Objective

To reason an underlying program from a set of demonstrations, the primary and essential step is recognizing actions and perceptions happening in each step of the demonstration. However, it can be difficult to learn meaningful representations purely from the sequence loss of programs when environments increase in visual complexity. To alleviate this issue, we propose to predict additional action sequences and a perception vector from the demonstrations as auxiliary tasks. The overview of the auxiliary tasks are illustrated in Figure 2.3.

The first auxiliary task is predicting action sequences. Given a demo vector encoded by the demonstration encoder, an action decoder LSTM decodes k th demo vector v_{demo}^k into a sequence of actions. During training, a sequential cross entropy loss similar to Equation 2.5 is optimized:

$$\mathcal{L}_A = -\frac{1}{MKT} \sum_{m=1}^M \sum_{k=1}^K \sum_{t=1}^T \log p(a_{m,t}^{k*} | A_{m,t-1}^k, v_{\text{demo}}^k), \quad (2.6)$$

where, $a_{m,t}^k$ is the t -th action token in k -th demonstration of m -th training example, $A_{m,t}^k = \{a_{m,1}^k, \dots, a_{m,t}^k\}$ is the history of previous actions at time step t .

The second auxiliary task is predicting perceptions for each frame of the demonstrations. We denote a perception vector $\Phi = \{\phi_1, \dots, \phi_L\} \in \{0, 1\}^L$ as a L dimension binary vector obtained by executing L

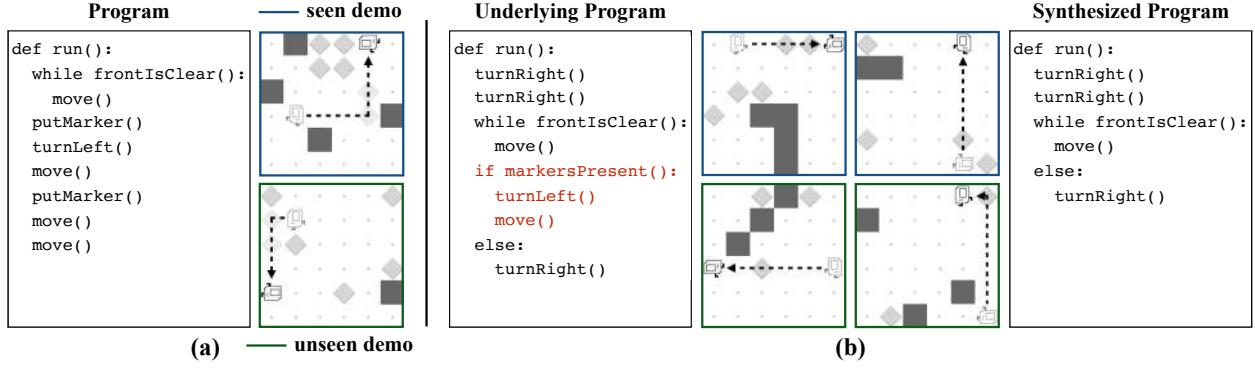


Figure 2.5: Karel Results. Seen training examples are on top row (in blue) and unseen testing examples are on the bottom row (in green). (a) A successful case with a program sequence match (b) Due to a missing branch condition execution in training data (top images), the synthesized program doesn't incorporate the condition, resulting in execution mismatch in lower right testing image.

perception primitives on a given state s . Specifically, we formulate the perception vector prediction as a sequential multi-label binary classification problem and optimizes the binary cross entropy.

$$\begin{aligned} \mathcal{L}_\Phi = & \\ - \frac{1}{MKTL} \sum_{m=1}^M \sum_{k=1}^K \sum_{t=1}^T \sum_{l=1}^L \log p(\phi_{m,t,l}^{k*} | P_{m,t-1}^k, v_{\text{demo}}^k), \end{aligned} \quad (2.7)$$

where $P_{m,t}^k = \{f(\Phi_{m,1}^k), \dots, f(\Phi_{m,t}^k)\}$ is the history of encoded previous perception vectors and $f(\cdot)$ is an encoding function.

The aggregated multi-task objective is as follows: $\mathcal{L} = \mathcal{L}_C + \alpha \mathcal{L}_A + \beta \mathcal{L}_\Phi$, where α and β are hyper-parameter controlling the importance of each loss. For all the experiments in this paper, we set $\alpha = \beta = 1$.

2.5 Experiments

We perform experiments in different environments: Karel [226] and ViZDoom [137]. We first describe the experimental setup and then present the experimental results.

2.5.1 Evaluation Metric

To verify whether a model is able to infer an underlying program η^* from a given set of demonstrations D , we evaluate accuracy based on the synthesized codes and the underlying program (*sequence accuracy* and *program accuracy*) as well as the execution of the program (*execution accuracy*).

Sequence accuracy Comparison in the code space is based on the instantiated code C^* of a ground truth program and the synthesized code \hat{C} from a program synthesizer. The *sequence accuracy* counts exact match of two code sequences, which is formally written as:

$$\text{Acc}_{\text{seq}} = \frac{1}{M} \sum_{m=1}^M \mathbb{1}_{\text{seq}}(C_m^*, \hat{C}_m),$$

where M is the number of testing examples and $\mathbb{1}_{\text{seq}}(\cdot, \cdot)$ is the indicator function of exact sequence match.

Program accuracy While the *sequence accuracy* is simple, it is a pessimistic estimation of *program accuracy* since it does not consider *program aliasing* – different codes with identical program semantics (e.g. `repeat(2) : (move())` and `move() move()`). Therefore, we measure the *program accuracy* by enumerating variations of codes. Specifically, we exploit the syntax of DSL to identify variations: e.g. unfolding repeat statements, decomposing if-else statement into two if statements, etc. Formally, the *program accuracy* is $\text{Acc}_{\text{program}} = \frac{1}{M} \sum_{m=1}^M \mathbb{1}_{\text{prog}}(C_m^*, \hat{C}_m)$, where $\mathbb{1}_{\text{prog}}(C_m^*, \hat{C}_m)$ is an indicator function that returns 1 if any variations of \hat{C}_m match any variations of C_m^* . Note that the *program accuracy* is only computable when the DSL is relatively simple and some assumptions are made *i.e.* termination of loops. The details of computing program accuracy are presented in Appendix (Section 2.7).

Execution accuracy To evaluate how well a synthesized program can capture the behaviors of an underlying program, we compare the execution results of the synthesized program code \hat{C} and the demonstrations D^* generated by a ground truth program η^* , where both are generated from the same set of sampled initial states $I_K = \{s_1^1, \dots, s_1^K\}$. We formally define the *execution accuracy* as: $\text{Acc}_{\text{execution}} =$

$\frac{1}{M} \sum_{m=1}^M \mathbb{1}_{\text{execution}}(D_m^*, \hat{D}_m)$, where $\mathbb{1}_{\text{execution}}(D_m^*, \hat{D}_m)$ is the indicator function of exact sequence match.

Note that when the number of sampled initial states becomes infinitely large, the *execution accuracy* converges to the *program accuracy*.

2.5.2 Evaluation Setting

For training and evaluation, we collect M_{train} training programs and M_{test} test programs. Each program code C_m^* is randomly sampled from an environment specific DSL and compiled into an executable form η_m^* . The corresponding demonstrations $D_m^* = \{\tau_1, \dots, \tau_K\}$ are generated by running the program on $K = K_{\text{seen}} + K_{\text{unseen}}$ different initial states. The seen demonstrations are used as an input to the program synthesizer, and the unseen demonstrations are used for computing execution accuracy. We train our model on the training set $\Omega_{\text{train}} = \{(C_1^*, D_1^*), \dots, (C_{M_{\text{train}}}^*, D_{M_{\text{train}}}^*)\}$ and test them on the testing set $\Omega_{\text{test}} = \{(C_1^*, D_1^*), \dots, (C_{M_{\text{test}}}^*, D_{M_{\text{test}}}^*)\}$. Note that Ω_{train} and Ω_{test} are disjoint. Both sequence and execution accuracies are used for the evaluation. The training details are described in Appendix (Section 2.7).

2.5.3 Baselines

We compare our proposed model (*ours*) against baselines to evaluate the effectiveness of: (1) explicitly modeling the underlying programs (2) our proposed model with the summarizer module and multi-task objective. To address (1), we design a program *induction baseline* based on [69], which bypasses synthesizing programs and directly predicts action sequences. We modified the architecture to incorporate multiple demonstrations as well as pixel inputs. The details are presented in Appendix (Section 2.7). For a fair comparison with our model that gets supervision of perception primitives, we feed the perception primitive vector of every frame as an input to the *induction baseline*. To verify (2), we compose a program *synthesis baseline* simply consisting of a demonstration encoder and a program decoder without a summarizer module

and multi-task loss. To integrate all the demonstration encoder outputs across demos, an average pooling layer is applied.

2.5.4 Karel

We first focus on a visually simple environment to verify the feasibility of program synthesis from demonstrations. We consider Karel [226] featuring an agent navigating through a gridworld with walls and interacting with markers based on the underlying program.

2.5.4.1 Environment and Dataset

Karel has 5 action primitives for moving and interacting with markers and 5 perception primitives for detecting obstacles and markers. A gridworld of 8×8 size is used for our experiments. To evaluate the generalization ability of the program synthesizer to novel programs, we randomly generate 35,000 unique programs and split them into a training set with 25,000 program, a validation set with 5,000 program, and a testing set with 5,000 programs. The maximum length of the program codes is 43. For each program, 10 seen demonstrations and 5 unseen demonstrations are generated. The maximum length of the demonstrations is 20.

Methods	Execution	Program	Se-	quence
Induction baseline	62.8% (69.1%)	-	-	
Synthesis baseline	64.1%	42.4%	35.7%	
+ summarizer (ours)	68.6%	45.3%	38.3%	
+ multi-task loss	72.1%	48.9%	41.0%	
(ours-full)				

Table 2.1: Performance evaluation on Karel environment. *Synthesis baseline* outperforms *induction baseline*. The summarizer module and the multi-task objective introduce significant improvement.

2.5.4.2 Performance Evaluation

The evaluation results of our proposed model and baselines are shown in Table 2.1. Comparison of execution accuracy shows relative performance of the proposed model and the baselines. *Synthesis baseline* outperforms *induction baseline* based on the execution accuracy, which shows the advantage of explicit modeling the underlying programs. *Induction baseline* often matches some of the K_{unseen} demonstration, but fails to match all of them from a single program. This observation is supported by the number in the parenthesis (69.1%), which counts the number of correct demonstrations while execution accuracy counts the number of program whose demonstrations match perfectly. This finding has also been reported in [63].

The proposed model shows consistent improvement over *synthesis baseline* for all the evaluation metrics. The sequence accuracy for our full model is 41.0%, which is a reasonable generalization performance given that none of the test programs are seen during training. We observe that our model often synthesizes programs that do not exactly match with the ground truth program but are semantically identical. For example, given a ground truth program `repeat(4) : (turnLeft; turnLeft; turnLeft)`, our model predicts `repeat (12) : (turnLeft)`. These cases are considered correct for program accuracy. Note that comparison based on the execution and sequence accuracy is consistent with the program accuracy, which justifies using them as a proxy for the program accuracy when it is not computable.

The qualitative success and failure cases of the proposed model are described in Figure 2.5. The Figure 2.5(a) shows a correct case where a single program is used to generate diverse action sequences. Figure 2.5(b) show a failure case, where part of the ground truth program tokens are not generated due to missing seen demonstration hitting that condition.

Methods	k=3	k=5	k=10
Synthesis baseline	58.5%	60.1%	64.1%
+ summarizer	60.6%	63.1%	68.6%
(ours)			
Improvement	2.1%	3.0%	4.5%

Table 2.2: Effect of the summarizer module. Employing the proposed summarizer module brings more improvement as the number of seen demonstration increases over *synthesis baseline*.

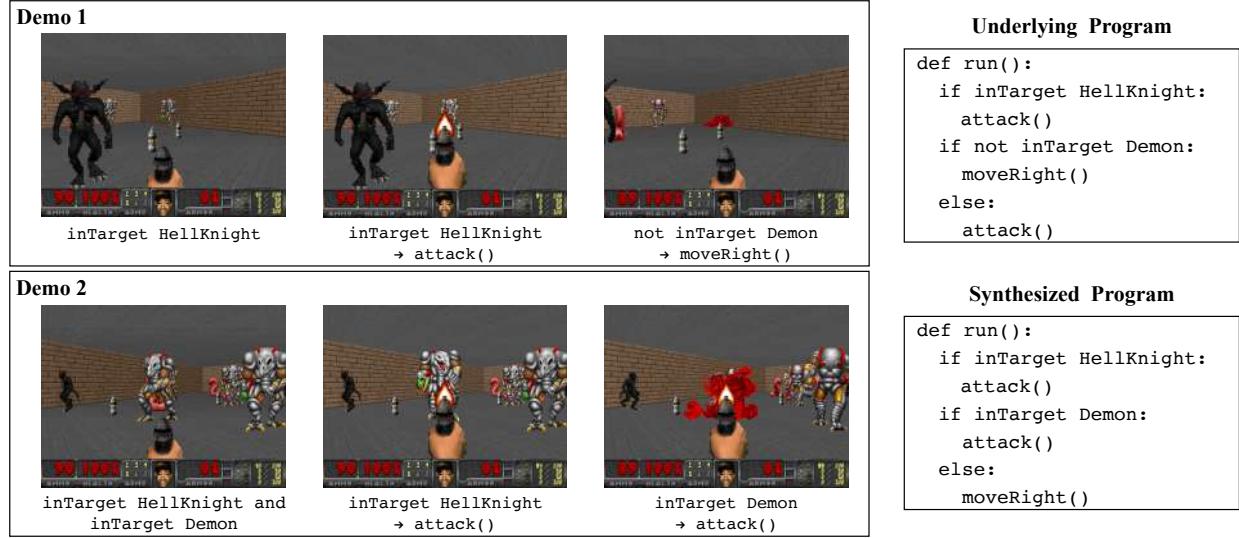


Figure 2.6: ViZDoom Results. Annotations below frames are the perception conditions and actions. Hellknight, Revenant, and Demon monsters are white, black, and pink respectively. The model is able to correctly percepts the condition and actions as well as synthesize a precise program. Note that the synthesized and the underlying program are semantically identical.

2.5.4.3 Effect of Summarizer

To verify the effectiveness of our proposed summarizer module, we conduct experiments where models are trained on varying numbers of demonstrations and compare the execution accuracy in Table 2.2. As the number of demonstrations increases, both models enjoy a performance gain due to extra available information. However, the gap between our proposed model and *synthesis baseline* also grows, which demonstrates the effectiveness of our summarizer module.

2.5.5 ViZDoom

Doom is a 3D first-person shooter game where a player can move in a continuous space and interact with monsters, items and weapons. We use ViZDoom [137], an open-source Doom-based AI platform, for our experiments. ViZDoom's increased visual complexity and a richer DSL could test the boundary of models in state comprehension, demo summarization, and program synthesis.

2.5.5.1 Environment and Dataset

The ViZDoom environment has 7 action primitives including diverse motions and attack as well as 6 perception primitives checking the existence of different monsters and whether they are targeted. Each state is represented by an image with $120 \times 160 \times 3$ pixels. For each demonstration, initial state is sampled by randomly spawning different types of monsters and ammos in different location and placing an agent randomly. To ensure that the program behavior results in the same execution, we control the environment to be deterministic.

We generate 80,000 training programs and 8,000 testing programs. To encourage diverse behavior of generated program, we give a higher sampling rate to the perception primitives that has higher entropy over K different initial states. We use 25 seen demonstrations for program synthesis and 10 unseen demonstrations for execution accuracy measure. The maximum length of programs is 32 and the maximum length of demonstrations is 20.

2.5.5.2 Performance Evaluation

Table 2.3 shows the result on ViZDoom environment. *Synthesis baseline* outperforms *induction baseline* in terms of the execution accuracy, which shows the strength of program synthesis for understanding diverse demonstrations. In addition, the proposed summarizer module and the multi-task objective bring improvement in terms of all evaluation metrics. Also we found that the syntax of the synthesized programs is about 99.9% accurate. This tells that the program synthesizer correctly learn the syntax of the DSL.

Figure 2.6 shows the qualitative result. It is shown that the generated program covers different conditional behavior in the demonstration successfully. In the example, the synthesized program does not match the underlying program in the code space, while matching the underlying program in the program space.

Methods	Execution	Program	Sequence
Induction baseline	35.1% (60.6%)	-	-
Synthesis baseline	48.2%	39.9%	33.1%
Ours-full	78.4%	62.5%	53.2%

Table 2.3: Performance evaluation on ViZDoom environment. The proposed model outperforms *induction baseline* and *synthesis baseline* significantly as the environment is more visually complex.

Methods	Execution	Program	Sequence
Induction baseline	26.5% (83.1%)	-	-
Synthesis baseline	59.9%	44.4%	36.1%
Ours-full	89.4%	69.1%	58.8%

Table 2.4: If-else experiment on ViZDoom environment. Single if-else statement with two branching consequences is used to evaluate ability of inferring underlying conditions.

2.5.5.3 Analysis

To verify the importance of inferring underlying conditions, we perform evaluation only with programs containing a single if-else statement with two branching consequences. This setting is sufficiently simple to isolate other diverse factors that might affect the evaluation result. For the experiment, we use 25 seen demonstrations to understand a behavior and 10 unseen demonstrations for testing. The result is shown in Table 2.4. *Induction baseline* has difficulty inferring the underlying condition to match all unseen demonstrations most of the times. In addition, our proposed model outperforms *synthesis baseline*, which demonstrates the effectiveness of the summarizer module and the multi-task objective.

Figure 2.7 illustrates how models trained with a fixed number (25) of seen demonstration generalize to fewer or more seen demonstrations during testing time. This shows our model and *synthesis baseline* are able to leverage more seen demonstrations to synthesize more accurate programs as well as achieve reasonable performance when fewer demonstrations are given. On the contrary, *Induction baseline* could not exploit more than 10 demonstrations well.

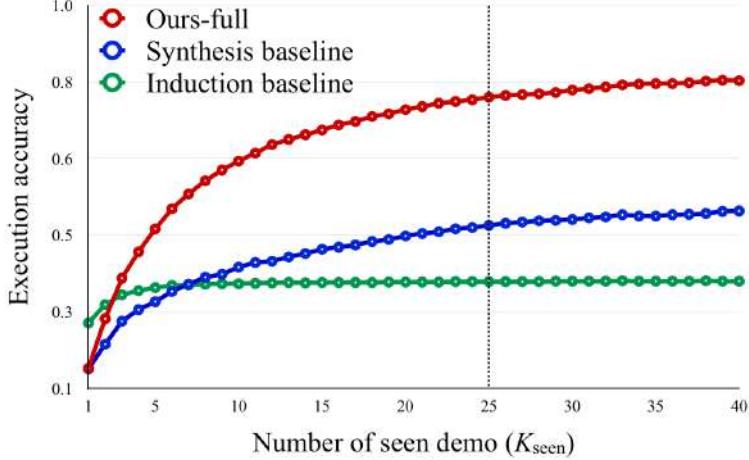


Figure 2.7: Generalization over different number of K_{seen} . The baseline models and our model trained with 25 seen demonstration are evaluated with fewer or more seen demonstrations.

2.5.5.4 Debugging the Synthesized Program

One of the intriguing properties of the program synthesis is that synthesized programs are interpretable and interactable by human. This makes it possible to debug a synthesized program and fix minor mistakes to correct the behaviors. To verify this idea, we use edit distance between synthesized program and ground truth program as a number of minimum token that is required to get a exactly correct program. With this setting, we found that fixing at most 2 program token provides 4.9% improvement in sequence accuracy and 4.1% improvement in execution accuracy.

2.6 Conclusion

We propose the task of synthesizing a program from diverse demonstration videos. To address this, we introduce a model augmented with a summarizer module to deal with branching conditions and a multi-task objective to induce meaningful latent representations. Our method is evaluated on a fully observable, third-person environment (Karel environment) and a partially observable, egocentric game (ViZDoom environment). The experiments demonstrate that the proposed model is able to reliably infer underlying programs and achieve satisfactory performances.

2.7 Appendix

2.7.1 Detailed Network Architectures

2.7.1.1 Demonstration Encoder

The demonstration encoder consists of a stack of convolutional layers and an LSTM. The stack of convolutional layers consists of five layers, which can be represented as:

$$C_{\{3,2,16\}} \rightarrow C_{\{3,2,32\}} \rightarrow C_{\{3,2,48\}} \rightarrow C_{\{3,2,48\}} \rightarrow C_{\{3,2,48\}},$$

where $C_{k,s,n}$ denotes a convolutional layer with a kernel size k , stride s , and a number of channel n . Then the encoded feature maps are flatten and passed to an LSTM. We experiment with RNN, GRU, and LSTM and found that LSTM works the best.

2.7.1.2 Summarizer Module

For the relation network of the summarizer module, we use two fully-connected layers with LeakyReLU activation. We also experiment with RNN, GRU, and LSTM for summarizer module and found that LSTM works the best.

2.7.1.3 Program Decoder

For the token embedding function used to produce embedding vectors of program tokens, we create an embedding lookup with a hidden size of 128. An LSTM with a hidden size of 512 is utilized to decode program tokens.

2.7.2 Training Details

We implement the proposed model and its submodules described in the main paper in TensorFlow [1] and trained it using batch size of 32 with Adam optimizer [140].

2.7.3 One-shot Imitation Learning Baseline

To evaluate the effectiveness of our proposed model, we implement an One-shot imitation learning model proposed in [69]. Since the model proposed in the original paper is not able to

1. Incorporate multiple seen specification demonstration sequences
2. Handle a varying-length number of demonstrations
3. Deal with visual input

we make modifications as follows:

1. Augment the demonstration encoder with a stack of convolutional layers to process visual input
2. Remove temporal dropout, temporal convolution, and neighborhood attention
3. Add an LSTM with an attention mechanism [178]. We also experimented with the monotonic attention mechanism [241] and empirically found [178] works better.
4. Replace the context network with an average pooling layer to handle a varying-length number of demonstrations
5. Change the manipulation network to an LSTM decoder, which optimizes the predictions of one-hot action vectors at each time step

2.7.4 Dataset Details

2.7.4.1 Karel

We use 5 action primitives and 5 perception primitives for Karel, which is formally defined as follows:

action := move | turnRight | turnLeft | pickMarker

| putMarker

perception := frontIsClear | leftIsClear | rightIsClear

| markersPresent | noMarkersPresent

For Karel environment, we use $8 \times 8 \times 16$ state representation, where each channel of the state representation has its own meaning.

0 : agent facing north | 1 : agent facing south ||

2 : agent facing west | 3 : agent facing east |

4 : wall or empty | 5 ~ 15 : 0 ~ 10 markers

2.7.4.2 ViZDoom

ViZDoom model contains 7 action primitives and 6 perception primitives, which is formally defined as follows:

action := moveBackward | moveForward | moveLeft

| moveRight | turnLeft | turnRight | attack

perception := isThere m | inTarget m

monster m := demon | hellKnight | revenant

ViZDoom environment has $120 \times 160 \times 3$ image as a state representation. We resize them to $80 \times 80 \times 3$ to feed to our model as an input.

To generate meaningful program and collecting diverse behavior we use heuristics to sample codes and demonstrations. Given each state we sequentially increase the program length by adding more statement. At the same time action is instantly taken to the environment and state transition is performed. Whenever statement with condition is sampled for the program, we give higher sampling probability to perception that makes current state more diverse.

Chapter 3

Learning to Synthesize Programs from Reward Functions

3.1 Introduction

Recently, deep reinforcement learning (DRL) methods have demonstrated encouraging performance on a variety of domains such as outperforming humans in complex games [200, 276, 277, 309] or controlling robots [39, 98, 15, 105, 333, 347, 159]. Despite the recent progress in the field, acquiring complex skills through trial and error still remains challenging and these neural network policies often have difficulty generalizing to novel scenarios. Moreover, such policies are not interpretable to humans and therefore are difficult to debug when these challenges arise.

To address these issues, a growing body of work aims to learn programmatic policies that are structured in more interpretable and generalizable representations such as decision trees [25], state-machines [123], and programs described by domain-specific programming languages [307, 306]. Yet, the programmatic representations employed in these works are often limited in expressiveness due to constraints on the policy spaces. For example, decision tree policies are incapable of naïvely generating repetitive behaviors, state machine policies used in [123] are computationally complex to scale to policies representing diverse behaviors, and the programs of [307, 306] are constrained to a set of predefined program templates. On

the other hand, program synthesis works that aim to represent desired behaviors using flexible domain-specific programs often require extra supervision such as input/output pairs [63, 35, 46, 273, 155] or expert demonstrations [286, 55], which can be difficult to obtain.

In this paper, we present a framework to instead synthesize human-readable programs in an expressive representation, solely from rewards, to solve tasks described by Markov Decision Processes (MDPs). Specifically, we represent a policy using a program composed of control flows (*e.g.* if/else and loops) and an agent’s perceptions and actions. Our programs can flexibly compose behaviors through perception-conditioned loops and nested conditional statements. However, composing individual program tokens (*e.g.* if, while, move()) in a trial-and-error fashion to synthesize programs that can solve given MDPs can be extremely difficult and inefficient.

To address this problem, we propose to first learn a latent program embedding space where nearby latent programs correspond to similar behaviors and allows for smooth interpolation, together with a program decoder that can decode a latent program to a program consisting of a sequence of program tokens. Then, when a task is given, this embedding space allows us to iteratively search over candidate latent programs to find a program that induces desired behavior to maximize the reward. Specifically, this embedding space is learned through reconstruction of randomly generated programs and the behaviors they induce in the environment in an unsupervised manner. Once learned, the embedding space can be reused to solve different tasks without retraining.

To evaluate the proposed framework, we consider the Karel domain [226], featuring an agent navigating through a gridworld and interacting with objects to solve tasks such as stacking and navigation. The experimental results demonstrate that the proposed framework not only learns to reliably synthesize task-solving programs but also outperforms program synthesis and deep RL baselines. In addition, we justify the necessity of the proposed two-stage learning scheme as well as conduct an extensive analysis comparing various approaches for learning the latent program embedding spaces. Finally, we perform

experiments which highlight that the programs produced by our proposed framework can both generalize to larger state spaces and unseen state configurations as well as be interpreted and edited by humans to improve their task performance.

3.2 Related Work

Neural program induction and synthesis. Program induction methods [155, 331, 61, 208, 95, 132, 89, 247, 38, 328, 37, 334, 168, 120] aim to implicitly induce the underlying programs to mimic the behaviors demonstrated in given task specifications such as input/output pairs or expert demonstrations. On the other hand, program synthesis methods [63, 35, 46, 273, 286, 31, 219, 273, 175, 171, 169, 75, 76, 24, 173, 3, 117, 278, 327, 44, 5, 48, 18, 117, 326, 47, 215] explicitly synthesize the underlying programs and execute the programs to perform the tasks from task specifications such input/output pairs, demonstrations, language instructions. In contrast, we aim to learn to synthesize programs solely from reward described by an MDP without other task specifications. Similarly to us, a two-stage synthesis method is proposed in [173]. Yet, the task is to match truth tables for given test programs rather than solve MDPs. Their first stage requires the entire ground-truth table for each program synthesized during training, which is infeasible to apply to our problem setup (*i.e.* synthesizing imperative programs for solving MDPs).

Learning programmatic policies. Prior works have also addressed the problem of learning programmatic policies [52, 325, 154]. Bastani, Pu, and Solar-Lezama [25] learns a decision tree as a programmatic policy for pong and cartpole environments by imitating an oracle neural policy. However, decision trees are incapable of representing repeating behaviors on their own. Silver et al. [278] addresses this by including a loop-style token for their decision tree policy, though it is still not as expressive as synthesized loops. Inala et al. [123] learns programmatic policies as finite state machines by imitating a teacher policy, although finite state machine complexity can scale quadratically with the number of states, making them difficult to scale to more complex behaviors.

```

Program  $\rho$  := DEF run m(  $s$  m)
Repetition  $n$  := Number of repetitions
Perception  $h$  := Domain-dependent perceptions
Condition  $b$  := perception  $h$  | not perception  $h$ 
Action  $a$  := Domain-dependent actions
Statement  $s$  := while c(  $b$  c) w(  $s$  w) |  $s_1; s_2$  |  $a$  |
repeat R= $n$  r(  $s$  r) | if c(  $b$  c) i(  $s$  i) |
ifelse c(  $b$  c) i(  $s_1$  i) else e(  $s_2$  e)

```

Figure 3.1: The domain specific language (DSL) for constructing programs.

Another line of work instead synthesizes programs structured in Domain-Specific Languages (DSLs), allowing humans to design tokens (*e.g.* conditions and operations) and control flows (*e.g.* while loops, if statements, reusable functions) to induce desired behaviors and can produce human interpretable programs. Verma et al. [307, 306] distill neural network policies into programmatic policies. Yet, the initial programs are constrained to a set of predefined program templates. This significantly limits the scope of synthesizable programs and requires designing such templates for each task. In contrast, our method can synthesize diverse programs, without templates, which can flexibly represent the complex behaviors required to solve various tasks.

3.3 Problem Formulation

We are interested in learning to synthesize a program structured in a given DSL that can be executed to solve a given task described by an MDP, purely from reward. In this section, we formally define our definition of a program and DSL, tasks described by MDPs, and the problem formulation.

Program and Domain-Specific Language. The programs, or programmatic policies, considered in this work are defined based on a DSL as shown in Figure 3.1. The DSL consists of control flows and an agent's

perceptions and actions. A perception indicates circumstances in the environment (e.g. `frontIsClear()`) that can be perceived by an agent, while an action defines a certain behavior that can be performed by an agent (e.g. `move()`, `turnLeft()`). Control flow includes `if/else` statements, loops, and boolean/logical operators to compose more sophisticated conditions. A policy considered in this work is described by a program ρ which is executed to produce a sequence of actions given perceptions from the environment.

MDP. We consider finite-horizon discounted MDPs with initial state distribution $\mu(s_0)$ and discount factor γ . For a fixed sequence $\{(s_0, a_0), \dots, (s_t, a_t)\}$ of states and actions obtained from a rollout of a given policy, the performance of the policy is evaluated based on a discounted return $\sum_{t=0}^T \gamma^t r_t$, where T is the horizon of the episode and $r_t = \mathcal{R}(s_t, a_t)$ the reward function.

Objective. Our objective is $\max_{\rho} \mathbb{E}_{a \sim \text{EXEC}(\rho), s_0 \sim \mu} [\sum_{t=0}^T \gamma^t r_t]$, where `EXEC` returns the actions induced by executing a program policy ρ in the environment. Note that one can view this objective as a special case of the standard RL objective, where the policy is represented as a program which follows the grammar of the DSL and the policy rollout is obtained by executing the program.

3.4 Approach

Our goal is to develop a framework that can synthesize a program (*i.e.* a programmatic policy) structured in a given DSL that can be executed to solve a task of interest. This requires the ability to synthesize a program that is not only valid for execution (e.g. grammatically correct) but also describes desired behaviors for solving the task from only the reward. Yet, learning to synthesize such a program from scratch for every new task can be difficult and inefficient.

To this end, we propose our **Learning Embeddings for Latent Program Synthesis** framework, dubbed LEAPS, as illustrated in Figure 3.2. LEAPS first learns a latent program embedding space that continuously parameterizes diverse behaviors and a program decoder that decodes a latent program to a program consisting of a sequence of program tokens. Then, when a task is given, we iteratively search over this

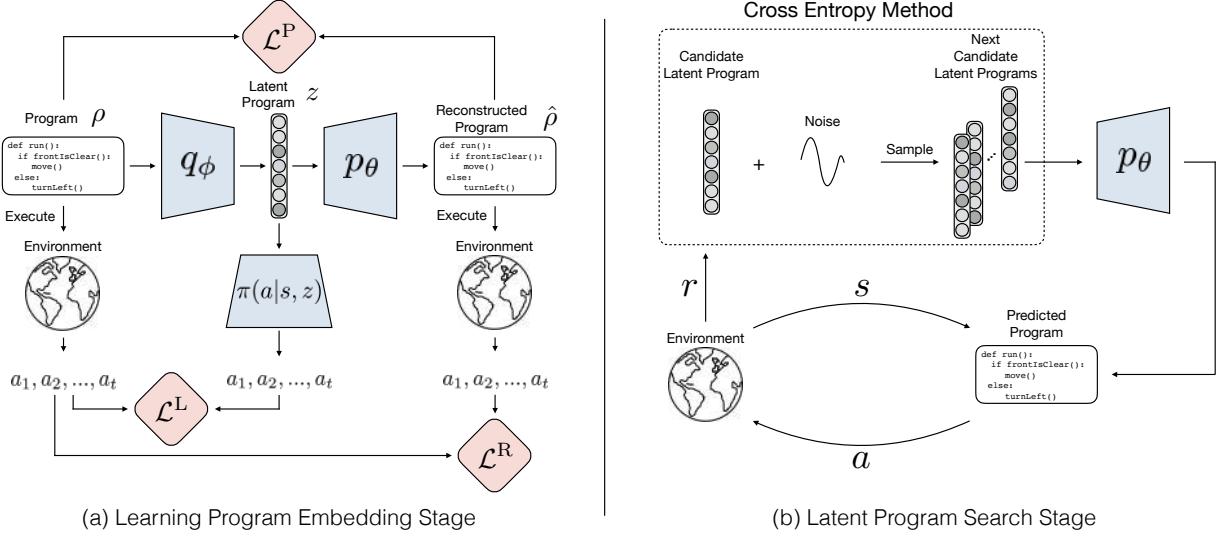


Figure 3.2: (a) **Learning program embedding stage**: we propose to learn a program embedding space by training a program encoder q_ϕ that encodes a program as a latent program z , a program decoder p_θ that decodes the latent program z back to a reconstructed program $\hat{\rho}$, and a policy π that conditions on the latent program z and acts as a neural program executor to produce the execution trace of the latent program z . The model optimizes a combination of a program reconstruction loss \mathcal{L}^P , a program behavior reconstruction loss \mathcal{L}^R , and a latent behavior reconstruction loss \mathcal{L}^L . a_1, a_2, \dots, a_t denotes actions produced by either the policy π or program execution. (b) **Latent program search stage**: we use the Cross Entropy Method to iteratively search for the best candidate latent programs that can be decoded and executed to maximize the reward to solve given tasks.

embedding space and decode each candidate latent program using the decoder to find a program that maximizes the reward. This two-stage learning scheme not only enables learning to synthesize programs to acquire desired behaviors described by MDPs solely from reward, but also allows reusing the learned embedding space to solve different tasks without retraining.

In the rest of this section, we describe how we construct the model and our learning objectives for the latent program embedding space in Section 3.4.1. Then, we present how a program that describes desired behaviors for a given task can be found through a search algorithm in Section 3.4.2.

3.4.1 Learning a Program Embedding Space

To learn a latent program embedding space, we propose to train a variational autoencoder (VAE) [141] that consists of a program encoder q_ϕ which encodes a program ρ to a latent program z and a program decoder

p_θ which reconstructs the program from the latent. Specifically, the VAE is trained through reconstruction of randomly generated programs and the behaviors they induce in the environment in an unsupervised manner. Architectural details are listed in Section 3.7.12.6.

Since we aim to iteratively search over the learned embedding space to achieve certain behaviors when a task is given, we want this embedding space to allow for smooth behavior interpolation (*i.e.* programs that exhibit similar behaviors are encoded closer in the embedding space). To this end, we propose to train the model by optimizing the following three objectives.

3.4.1.1 Program Reconstruction

To learn a program embedding space, we train a program encoder q_ϕ and a program decoder p_θ to reconstruct programs composed of sequences of program tokens. Given an input program ρ consisting of a sequence of program tokens, the encoder processes the input program one token at a time and produces a latent program embedding z . Then, the decoder outputs program tokens one by one from the latent program embedding z to synthesize a reconstructed program $\hat{\rho}$. Both the encoder and the decoder are recurrent neural networks and are trained to optimize the β -VAE [113] loss:

$$\mathcal{L}_{\theta,\phi}^P(\rho) = -\mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\rho)}[\log p_\theta(\rho|\mathbf{z})] + \beta D_{\text{KL}}(q_\phi(\mathbf{z}|\rho) \| p_\theta(\mathbf{z})). \quad (3.1)$$

3.4.1.2 Program Behavior Reconstruction

While the loss in Equation 3.1 enforces that the model encodes syntactically similar programs close to each other in the embedding space, we also want to encourage programs with the same semantics to have similar program embeddings. An example that demonstrates the importance of this is the *program aliasing* issue, where different programs have identical program semantics (*e.g.* `repeat(2): move()` and `move() move()`). Thus, we introduce an objective that compares the execution traces of the input program and the

reconstructed program. Since the program execution process is not differentiable, we optimize the model via REINFORCE [324]:

$$\mathcal{L}_{\theta, \phi}^R(\rho) = -\mathbb{E}_{z \sim q_\phi(z|\rho)}[R_{\text{mat}}(p_\theta(\rho|z), \rho)], \quad (3.2)$$

where $R_{\text{mat}}(\hat{\rho}, \rho)$, the reward for matching the input program's behavior, is defined as

$$R_{\text{mat}}(\hat{\rho}, \rho) = \mathbb{E}_\mu[\frac{1}{N} \sum_{t=1}^N \underbrace{\mathbb{1}\{\text{EXEC}_i(\hat{\rho}) == \text{EXEC}_i(\rho) \forall i = 1, 2, \dots, t\}}_{\text{stays 0 after the first } t \text{ where } \text{EXEC}_t(\hat{\rho}) != \text{EXEC}_t(\rho)}], \quad (3.3)$$

where N is the maximum of the lengths of the execution traces of both programs, and $\text{EXEC}_i(\rho)$ represents the action taken by program ρ at time i . Thus this objective encourages the model to embed behaviorally similar yet possibly syntactically different programs to similar latent programs.

3.4.1.3 Latent Behavior Reconstruction

To further encourage learning a program embedding space that allows for smooth behavior interpolation, we devise another source of supervision by learning a program embedding-conditioned policy. Denoted $\pi(a|z, s_t)$, this recurrent policy takes the program embedding z produced by the program encoder and learns to predict corresponding agent actions. One can view this policy as a neural program executor that allows gradient propagation through the policy and the program encoder by optimizing the cross entropy between the actions obtained by executing the input program ρ and the actions predicted by the policy:

$$\mathcal{L}_\pi^L(\rho, \pi) = -\mathbb{E}_\mu[\sum_{t=1}^M \sum_{i=1}^{|\mathcal{A}|} \mathbb{1}\{\text{EXEC}_i(\hat{\rho}) == \text{EXEC}_i(\rho)\} \log \pi(a_i|z, s_t)], \quad (3.4)$$

where M denotes the length of the execution of ρ . Optimizing this objective directly encourages the program embeddings, through supervised learning instead of RL as in \mathcal{L}^R , to be useful for action reconstruction, thus further ensuring that similar behaviors are encoded together and allowing for smooth interpolation.

Note that this policy is only used for improving learning the program embedding space not for solving the tasks of interest in the later stage.

In summary, we propose to optimize three sources of supervision to learn the program embedding space that allows for smooth interpolation and can be used to search for desired agent behaviors: (1) \mathcal{L}^P (Equation 3.1), the β -VAE objective for program reconstruction, (2) \mathcal{L}^R (Equation 3.2), an RL environment-state matching loss for the reconstructed program, and (3) \mathcal{L}^L (Equation 3.4), a supervised learning loss to encourage predicting the ground-truth agent action sequences. Thus our combined objective is:

$$\min_{\theta, \phi, \pi} \lambda_1 \mathcal{L}_{\theta, \phi}^P(\rho) + \lambda_2 \mathcal{L}_{\theta, \phi}^R(\rho) + \lambda_3 \mathcal{L}_{\pi}^L(\rho, \pi), \quad (3.5)$$

where λ_1 , λ_2 , and λ_3 are hyperparameters controlling the importance of each loss. Optimizing the combination of these losses encourages the program embedding to be both semantically and syntactically informative. More training details can be found in Section 3.7.12.6.

3.4.2 Latent Program Search: Synthesizing a Task-Solving Program

Once the program embedding space is learned, our goal becomes searching for a latent program that maximizes the reward described by a given task MDP. To this end, we adapt the Cross Entropy Method (CEM) [253], a gradient-free continuous search algorithm, to iteratively search over the program embedding space. Specifically, we (1) sample a distribution of latent programs, (2) decode the sampled latent programs into programs using the learned program decoder p_θ , (3) execute the programs in the task environment and obtain the corresponding rewards, and (4) update the CEM sampling distribution based on the rewards. This process is repeated until either convergence or the maximum number of sampling steps has been reached.

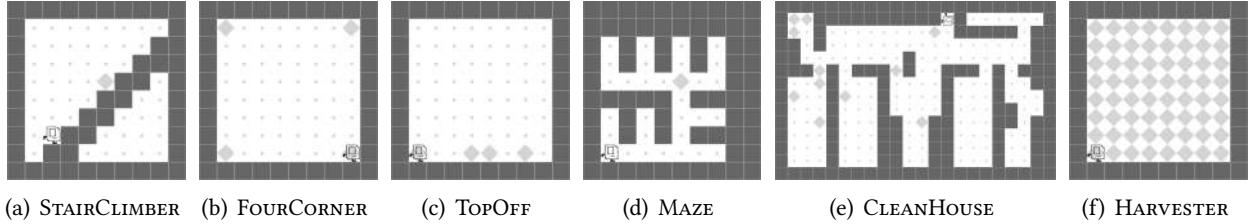


Figure 3.3: **The Karel problem set**: the domain features an agent navigating through a gridworld with walls and interacting with markers, allowing for designing tasks that demand certain behaviors. The tasks are further described in Section 3.7.11 with visualizations in Figure 3.18.

3.5 Experiments

We first introduce the environment and the tasks in Section 3.5.1 and describe the experimental setup in Section 3.5.2. Then, we justify the design of LEAPS by conducting extensive ablation studies in Section 3.5.3. We describe the baselines used for comparison in Section 3.5.4, followed by the experimental results presented in Section 3.5.5. In Section 3.5.6, we conduct experiments to evaluate the ability of our method to generalize to a larger state space without further learning. Finally, we investigate how LEAPS' interpretability can be leveraged by conducting experiments that allow humans to debug and improve the programs synthesized by LEAPS in Section 3.5.7

3.5.1 Karel Domain

To evaluate the proposed framework, we consider the Karel domain [226], as featured in [35, 273, 286], which features an agent navigating through a gridworld with walls and interacting with markers. The agent has 5 actions for moving and interacting with marker and 5 perceptions for detecting obstacles and markers. The tasks of interest are shown in Figure 3.3. Note that most tasks have randomly sampled agent, wall, marker, and/or goal configurations. When either training or evaluating, we randomly sample initial configurations upon every episode reset. More details can be found in Section 3.7.11.

3.5.2 Programs

To produce programs for learning the program embedding space, we randomly generated a dataset of 50,000 unique programs. Note that the programs are generated independently of any Karel tasks; each program is created only by sampling tokens from the DSL, similar to the procedures used in [63, 35, 46, 273, 286, 55]. This dataset is split into a training set with 35,000 programs a validation set with 7,500 programs, and a testing set with 7,500 programs. The validation set is used to select the learned program embedding space to use for the program synthesis stage.

For each program, we sample random Karel states and execute the program on them from different starting states to obtain 10 environment rollouts to compute the program behavior reconstruction loss \mathcal{L}^R and the latent behavior reconstruction loss \mathcal{L}^L when learning the program embedding space. We perform checks to ensure rollouts cover all execution branches in the program so that they are representative of all aspects of the program’s behavior. The maximum length of the programs is 44 tokens and the average length is 17.9. We plot a histogram of their lengths in Figure 3.17 (in Appendix). More dataset generation details can be found in Section 3.7.10.

3.5.3 Ablation Study

We first ablate various components of our proposed framework in order to (1) justify the necessity of the proposed two-stage learning scheme and (2) identify the effects of the proposed objectives. We consider the following baselines and ablations of our method (illustrated Section 3.7.9).

- Naïve: a program synthesis baseline that learns to directly synthesize a program from scratch by recurrently predicting a sequence of program tokens. This baseline investigates if an end-to-end learning method can solve the problem. More details can be found in Section 3.7.12.4.

- LEAPS-P: the simplest ablation of LEAPS, in which the program embedding space is learned by only optimizing the program reconstruction loss \mathcal{L}^P (Equation 3.1).
- LEAPS-P+R: an ablation of LEAPS which optimizes both the program reconstruction loss \mathcal{L}^P (Equation 3.1) and the program behavior reconstruction loss \mathcal{L}^R (Equation 3.2).
- LEAPS-P+L: an ablation of LEAPS which optimizes both the program reconstruction loss \mathcal{L}^P (Equation 3.1) and the latent behavior reconstruction loss \mathcal{L}^L (Equation 3.4).
- LEAPS (LEAPS-P+R+L): LEAPS with all the losses, optimizing our full objective in Equation 3.5.
- LEAPS-rand-{8/64}: similar to LEAPS, this ablation also optimizes the full objective (Equation 3.5) for learning the program embedding space. Yet, when searching latent programs, instead of CEM, it simply randomly samples 8/64 candidate latent programs and chooses the best performing one. These baselines justify the effectiveness of using CEM for searching latent programs.

Table 3.1: Program behavior reconstruction rewards (standard deviations) across all methods.

	WHILE	IFELSE+WHILE	2IF+IFELSE	WHILE+2IF+IFELSE	Avg Reward
Naïve	0.65 (0.33)	0.83 (0.07)	0.61 (0.33)	0.16 (0.06)	0.56
LEAPS-P	0.95 (0.13)	0.82 (0.08)	0.58 (0.35)	0.33 (0.17)	0.67
LEAPS-P+R	0.98 (0.09)	0.77 (0.05)	0.63 (0.25)	0.52 (0.27)	0.72
LEAPS-P+L	1.06 (0.00)	0.84 (0.10)	0.77 (0.23)	0.33 (0.13)	0.75
LEAPS-rand-8	0.62 (0.24)	0.49 (0.09)	0.36 (0.18)	0.28 (0.14)	0.44
LEAPS-rand-64	0.78 (0.22)	0.63 (0.09)	0.55 (0.20)	0.37 (0.09)	0.58
LEAPS	1.06 (0.08)	0.87 (0.13)	0.85 (0.30)	0.57 (0.23)	0.84

Program Behavior Reconstruction. To determine the effectiveness of the proposed two-stage learning scheme and the learning objectives, we measure how effective each ablation is at reconstructing the behaviors of input programs. We use programs from the test set (shown in Figure 3.11 in Appendix), and utilize the environment state matching reward $R_{\text{mat}}(\hat{\rho}, \rho)$ (Equation 3.3), with a 0.1 bonus for synthesizing a syntactically correct program. Thus the return ranges between [0, 1.1]. We report the mean cumulative return, over 5 random seeds, of the final programs after convergence.

The results are reported in Table 3.1. Each test is named after its control flows (e.g. IFELSE+WHILE has an if-else statement and a while loop). The naïve program synthesis baseline fails on the complex WHILE+2IF+IFELSE program, as it rarely synthesizes conditional and loop statements, instead generating long sequences of action tokens that attempt to replicate the desired behavior of those statements (see synthesized programs in Figure 3.12). We believe that this is because it is incentivized to initially predict action tokens to gain more immediate reward, making it less likely to synthesize other tokens. LEAPS and its variations perform better and synthesize more complex programs, demonstrating the importance of the proposed two-stage learning scheme in biasing program search. We also note that LEAPS-P achieves the worst performance out of the CEM search LEAPS ablations, indicating that optimizing the program reconstruction loss \mathcal{L}^P (the VAE loss) alone does not yield satisfactory results. Jointly optimizing \mathcal{L}^P with either the program behavior reconstruction loss \mathcal{L}^R or the latent behavior reconstruction loss \mathcal{L}^L improves the performance, and optimizing our full objective with all three achieves the best performance across all tasks, indicating the effectiveness of the proposed losses. Finally, LEAPS outperforms LEAPS-rand-8/64, suggesting the necessity of adopting better search algorithms such as CEM.

Program Embedding Space Smoothness.

We investigate if the program and latent behavior reconstruction losses encourage learning a behaviorally smooth embedding space. To quantify behavioral smoothness, we measure how much a change in the embedding space corresponds to a change in behavior by comparing execution traces. For all programs we compute the pairwise Euclidean dis-

Table 3.2: Program embedding space smoothness. For each program, we execute the ten nearest programs in the learned embedding space of each model to calculate the mean state-matching reward R_{mat} against the original program. We report R_{mat} averaged over all programs in each dataset.

	LEAPS-P	LEAPS-P+R	LEAPS-P+L	LEAPS
TRAINING	0.22	0.22	0.31	0.31
VALIDATION	0.22	0.21	0.27	0.27
TESTING	0.22	0.22	0.28	0.27

tance between their embeddings in each model. We then calculate the environment state matching distance R_{mat} between the decoded programs by executing them from the same initial state.

The results are reported in Table 3.2. LEAPS and LEAPS-P+L perform the best, suggesting that optimizing the latent behavior reconstruction objective \mathcal{L}^L , in Equation 3.4, is essential for improving the smoothness of the latent space in terms of execution behavior. We further analyze and visualize the learned program embedding space in Section 3.7.1 and Figure 3.4 (in Appendix).

3.5.4 Baselines

We evaluate LEAPS against the following baselines (illustrated in Figure 3.16 in Appendix Section 3.7.9).

- DRL: a neural network policy trained on each task and taking raw states (Karel grids) as input.
- DRL-abs: a recurrent neural network policy directly trained on each Karel task but taking *abstract* states as input (*i.e.* it sees the same perceptions as LEAPS, *e.g.* `frontIsClear() == true`).
- DRL-abs-t: a DRL transfer learning baseline in which for each task, we train DRL-abs policies on all other tasks, then fine-tune them on the current task. Thus it acquires a prior by learning to first solve other Karel tasks. Rewards are reported for the policies from the task that transferred with highest return. We only transfer DRL-abs policies as some tasks have different state spaces.
- HRL: a hierarchical RL baseline in which a VAE is first trained on action sequences from program execution traces used by LEAPS. Once trained, the decoder is utilized as a low-level policy for learning a high-level policy to sample actions from. Similar to LEAPS, this baseline utilizes the dataset to produce a prior of the domain. It takes raw states (Karel grids) as input.
- HRL-abs: the same method as HRL but taking abstract states (*i.e.* local perceptions) as input.
- VIPER [25]: A decision-tree programmatic policy which imitates the behavior of a deep RL teacher policy via a modified DAgger algorithm [251]. This decision tree policy cannot synthesize loops, allowing us to highlight the performance advantages of more expressive program representation that LEAPS is able to take advantage of.

Table 3.3: Mean return (standard deviation) of all methods across Karel tasks, evaluated over 5 random seeds. DRL methods, program synthesis baselines, and LEAPS ablations are separately grouped.

	STAIRCLIMBER	FOURCORNER	TOPOFF	MAZE	CLEANHOUSE	HARVESTER
DRL	1.00 (0.00)	0.29 (0.05)	0.32 (0.07)	1.00 (0.00)	0.00 (0.00)	0.90 (0.10)
DRL-abs	0.13 (0.29)	0.36 (0.44)	0.63 (0.23)	1.00 (0.00)	0.01 (0.02)	0.32 (0.18)
DRL-abs-t	0.00 (0.00)	0.05 (0.10)	0.17 (0.11)	1.00 (0.00)	0.01 (0.02)	0.16 (0.18)
HRL	-0.51 (0.17)	0.01 (0.00)	0.17 (0.11)	0.62 (0.05)	0.01 (0.00)	0.00 (0.00)
HRL-abs	-0.05 (0.07)	0.00 (0.00)	0.19 (0.12)	0.56 (0.03)	0.00 (0.00)	-0.03 (0.02)
Naïve	0.40 (0.49)	0.13 (0.15)	0.26 (0.27)	0.76 (0.43)	0.07 (0.09)	0.21 (0.25)
VIPER	0.02 (0.02)	0.40 (0.42)	0.30 (0.06)	0.69 (0.05)	0.00 (0.00)	0.51 (0.07)
LEAPS-rand-8	0.10 (0.17)	0.10 (0.14)	0.28 (0.05)	0.40 (0.50)	0.00 (0.00)	0.07 (0.06)
LEAPS-rand-64	0.18 (0.40)	0.20 (0.11)	0.33 (0.07)	0.58 (0.41)	0.03 (0.06)	0.12 (0.05)
LEAPS	1.00 (0.00)	0.45 (0.40)	0.81 (0.07)	1.00 (0.00)	0.18 (0.14)	0.45 (0.28)

All the baselines are trained with PPO [265] or SAC [104], including the VIPER teacher policy. More training details can be found in Section 3.7.12.

3.5.5 Results

We present the results of the baselines and our method evaluated on the Karel task set based on the environment rewards in Table 3.3. The reward functions are sparse for all tasks, and are normalized such that the final cumulative return is within $[-1, 1]$ for tasks with penalties and $[0, 1]$ for tasks without; reward functions for each task are detailed in Section 3.7.11.

Overall Task Performance. Across all but one task, LEAPS yields the best performance. The LEAPS-rand baselines perform significantly worse than LEAPS on all Karel tasks, demonstrating the need for using a search algorithm like CEM during synthesis. The performance of VIPER is bounded by its RL teacher policy, and therefore is outperformed by the DRL baselines on most of the tasks. Meanwhile, DRL-abs-t is generally unable to improve upon DRL-abs across the board, suggesting that transferring Karel behaviors with RL from one task to another is ineffective. Furthermore, both the HRL baselines achieve poor performance, likely because agent actions alone provide insufficient supervision for a VAE to encode useful action trajectories on unseen tasks—unlike programs. Finally, the poor performance of the naïve

program synthesis baseline highlights the difficulty and inefficiency of learning to synthesize programs from scratch using only rewards. In the appendix, we present programs synthesized by LEAPS in Figure 3.14, example optimal programs for each task in Section 3.7.6 (Figure 3.11), rollout visualizations in Figure 3.19, and additional results analysis in Section 3.7.8.

Repetitive Behaviors. Solving STAIRCLIMBER and FOURCORNER requires acquiring repetitive (or looping) behaviors. STAIRCLIMBER, which can be solved by repeating a short, 4-step stair-climbing behavior until the goal marker is reached, is not solved by DRL-abs. LEAPS fully solves the task given the same perceptions, as this behavior can be simply represented with a while loop that repeats the stair-climbing skill. However VIPER performs poorly as its decision tree cannot represent such loops. Similarly, the baselines are unable to perform as well on FOURCORNER, a task in which the agent must pickup a marker located in each corner of the grid. This behavior takes at least 14 timesteps to complete, but can be represented by two nested loops. Similar to STAIRCLIMBER, the bias introduced by the DSL and our generated dataset (which includes nested loops), results in LEAPS being able to perform much better.

Exploration. TOPOFF rewards the agent for adding markers to locations with existing markers. However, there are no restrictions for the agent to wander elsewhere around the environment, thus making exploration a problem for the RL baselines, and thereby also constraining VIPER. LEAPS performs best on this task, as the ground-truth program can be represented by a simple loop that just moves forward and places markers when a marker is detected. MAZE also involves exploration, however its small size (8×8) results in many methods, including LEAPS, solving the task.

Complexity. Solving HARVESTER and CLEANHOUSE requires acquiring complex behaviors, resulting in poor performance from all methods. CLEANHOUSE requires an agent to navigate through a house and pick up all markers along the walls on the way. This requires repeated execution of a skill, of varied length, which navigates around the house, turns into rooms, and picks up markers. As such, all baselines perform very poorly. However, LEAPS is able to perform substantially better because these behaviors

can be represented by a program of medium complexity with a while loop and some nested conditional statements. On the other hand, HARVESTER involves simply navigating to and picking up a marker on every spot on the grid. However, this is a difficult program to synthesize given our random dataset generation process; the program we manually derive to solve HARVESTER is long and more syntactically complex than most training programs. As a result, DRL and VIPER outperform LEAPS on this task.

Learned Program Embedding Space. More analysis on our learned program embedding space can be found in the appendix. We present CEM search trajectory visualizations in Section 3.7.2, demonstrating how the search population’s rewards change over time. To qualitatively investigate the smoothness of the learned program embedding space, we linearly interpolate between pairs of latent programs and display their corresponding decoded programs in Section 3.7.3. In Section 3.7.4, we illustrate how predicted programs evolve over the course of CEM search.

3.5.6 Generalization

We are also interested in learning whether the baselines and the programs synthesized by LEAPS can generalize to novel scenarios without further learning. Specifically, we investigate how well they can generalize to larger state spaces. We expand both STAIRCLIMBER and MAZE to 100×100 grid sizes (from 12×12 and 8×8 , respectively). We directly evaluate the policies or programs obtained from the original tasks with smaller state spaces for all methods except

DRL (its observation space changes), which we retrain from scratch. The results are shown in Table 3.4. All baselines perform significantly worse than before on both tasks. On the contrary, the programs synthesized by LEAPS for the smaller task instances achieve zero-shot generalization to larger task instances without

Table 3.4: Rewards on 100×100 grids.

	STAIRCLIMBER	MAZE
DRL	0.00 (0.00)	0.00 (0.00)
DRL-abs	0.00 (0.00)	0.04 (0.05)
VIPER	0.00 (0.00)	0.10 (0.12)
LEAPS	1.00 (0.00)	1.00 (0.00)

losing any performance. Larger grid size experiments for the other Karel tasks and additional unseen configuration experiments can be found in Section 3.7.7.

3.5.7 Interpretability

Interpretability in machine learning [172, 271] is particularly crucial when it comes to learning a policy that interacts with the environment [349, 112, 83, 106, 255, 29, 296, 17]. The proposed framework produces programmatic policies that are interpretable from the following aspects as outlined in [271].

- Trust: interpretable machine learning methods and models may more easily be trusted since humans tend to be reluctant to trust systems that they do not understand. Programs synthesized by LEAPS can naturally be better trusted since one can simply read and interpret them.
- Contestability: the program execution traces produce a chain of reasoning for each action, providing insights on the induced behaviors and thus allowing for contesting improper decisions.
- Safety: synthesizing readable programs allows for diagnosing issues earlier (*i.e.* before execution) and provides opportunities to intervene, which is especially critical for safety-critical tasks.

In the rest of this section, we investigate how the proposed framework enjoys interpretability from the three aforementioned aspects. Specifically, synthesized programs are not only readable to human users but also interactive, allowing non-expert users with a basic understanding of programming to diagnose and make edits to improve their performance. To demonstrate this, we asked non-expert humans to read, interpret, and edit suboptimal LEAPS policies to improve their performance. Participants edited LEAPS programs on 3 Karel tasks with suboptimal reward: `TOPOFF`, `FOURCORNER`, and `HARVESTER`. With just 3 edits, participants obtained a mean reward improvement of 97.1%, and with 5 edits, participants improved it by 125%. This justifies how our synthesized policies can be manually diagnosed and improved, a property which DRL methods lack. More details and discussion can be found in Section 3.7.5.

3.6 Discussion

We propose a framework for solving tasks described by MDPs by producing programmatic policies that are more interpretable and generalizable than neural network policies learned by deep reinforcement learning methods. Our proposed framework adopts a flexible program representation and requires only minimal supervision compared to prior programmatic reinforcement learning and program synthesis works. Our proposed two-stage learning scheme not only alleviates the difficulty of learning to synthesize programs from scratch but also enables reusing its learned program embedding space for various tasks. The experiments demonstrate that our proposed framework outperforms DRL and programmatic baselines on a set of Karel tasks by producing expressive and generalizable programs that can consistently solve the tasks. Ablation studies justify the necessity of the proposed two-stage learning scheme as well as the effectiveness of the proposed learning objectives.

While the proposed framework achieves promising results, we would like to acknowledge two assumptions that are implicitly made in this work. First, we assume the existence of a program executor that can produce execution traces of programs. This program executor needs to be able to return perceptions from the environment state as well as apply actions to the environment. While this assumption is widely made in program synthesis works, a program executor can still be difficult to obtain when it comes to real-world robotic tasks. Fortunately, in research fields such as computer vision or robotics, a great amount of effort has been put into satisfying this assumption such as designing modules that can return high-level abstraction of raw sensory input (*e.g.* with object detection networks, proximity/tactile sensors, etc.).

Secondly, we assume that it is possible to generate a distribution of programs whose behaviors are at least remotely related to the desired behaviors for solving the tasks of interest. It can be difficult to synthesize programs which represent behaviors that are more complex than ones in the training program distribution, although one possible solution is to employ a better program generation process to generate programs that induce more complex behaviors. Also, the choice of DSL plays an important role in how

complex the programs can be. Ideally, employing a more complex DSL would allow our proposed framework to synthesize more advanced agent behaviors.

In the future, we hope to extend the proposed framework to more challenging domains such real-world robotics. We believe this framework would allow for deploying robust, interpretable policies for safety-critical tasks such as robotic surgeries. One way to make LEAPS applicable to robotics domains would be to simultaneously learn perception modules and action controllers. Other possible solutions include incorporating program execution methods [11, 216, 287, 336, 160, 350] that are designed to allow program execution or designing DSLs that allow pre-training of perception modules and action controllers. Also, the proposed framework shares some characteristics with works in multi-task RL [216, 11, 294, 298, 281, 124, 232] and meta-learning [280, 314, 310, 77, 315, 45, 158, 212, 239, 49]. Specifically, it learns a program embedding space from a distribution of tasks/programs. Once the program embedding space is learned, it can be reused to solve different tasks without retraining.

Yet, extending LEAPS to such domains can potentially lead to some negative societal impacts. For example, our framework can still capture unintended bias during learning or suffer from adversarial attacks. Furthermore, policies deployed in the real world can create great economic impact by causing job losses in some sectors. Therefore, we would encourage further work to investigate the biases, safety issues, and potential economic impacts to ensure that the deployment in the field does not cause far-reaching, negative societal impacts.

3.7 Appendix

3.7.1 Program Embedding Space Visualizations

In this section, we present and analyze visualizations providing insights on the program embedding spaces learned by LEAPS and its variations. To investigate the learned program embedding space, we perform

dimensionality reduction with PCA [130] to embed the following data to a 2D space for visualizations shown in Figure 3.4:

- Latent programs from the training dataset encoded by a learned encoder q_ϕ , visualized as blue scatters.
There are 35k training programs.
- Samples drawn from a normal distribution $\mathcal{N}(0, 1)$, visualized as green scatters. This is to show how a distribution would look like if the embedding space is learned by using a highly weighted KL-divergence penalty (*i.e.* a large β value the VAE loss). We compared this against the latent program distribution learned by our method to justify the effectiveness of the proposed objectives: the program behavior reconstruction loss (\mathcal{L}^R) and the latent behavior reconstruction loss (\mathcal{L}^L).
- Ground-truth (GT) test programs from the testing dataset, encoded by a learned decoder q_ϕ , visualized as plus signs (+) with different colors. We selected 4 test programs.
- Reconstructed programs which are predicted (Pred) by each method given visualized as crosses (\times) with different colors. Since there are 4 test programs selected, 4 reconstructed programs are visualized. Each pair of test program and predicted program is visualized with the same color. These predicted (*i.e.* synthesized) programs are also shown in Figure 3.12.

Embedding Space Coverage. Even though the testing programs are not in the training program dataset, and therefore are unseen to models, their embedding vectors still lie in the distribution learned by all the models. This indicates that the learned embedding spaces cover a wide distribution of programs.

Latent Program Distribution vs. Normal Distribution. We now compare two distributions: the latent program distribution formed by encoding all the training programs to the program embedding space and a normal distribution $\mathcal{N}(0, 1)$. One can view the normal distribution as the distribution obtained by heavily enforcing the weight of the KL-divergence term when training a VAE model. We discuss the shape of the latent program distribution in the learned program embedding space as follows:

- LEAPS-P: since LEAPS+P simply optimizes the β -VAE loss (the program reconstruction loss \mathcal{L}^P), which puts a lot of emphasis on the KL-divergence term, the shape of the latent program distribution is very similar to a normal distribution as shown in Figure 3.4 (a).
- LEAPS-P+R: while LEAPS+P+R additionally optimizes the program behavior reconstruction loss \mathcal{L}^R , the shape of the latent program distribution is still similar to a normal distribution, as shown in Figure 3.4 (b). We hypothesize that it is because the program behavior reconstruction loss alone might not be strong or explicit enough to introduce a change.
- LEAPS-P+L: the shape of the latent program distribution in the program embedding space learned by LEAPS+P+L is significantly different from a normal distribution, as shown in Figure 3.4 (c). This suggest that employing the latent behavior reconstruction loss \mathcal{L}^L dramatically contributes to the learning. We believe it is because the latent behavior reconstruction loss is optimized with direct gradients and therefore provides a stronger learning signal especially compared to the program behavior reconstruction loss \mathcal{L}^R , which is optimized using REINFORCE [324].
- LEAPS (LEAPS-P+R+L): LEAPS optimizes the full objective that includes all three proposed objectives and form a similar distribution shape as the one learned by LEAPS+P+L. However, the distance between each pair of the ground-truth testing program and the predicted program is much closer in the program embedding space learned by LEAPS compared to the space learned by LEAPS+P+L. This justifies the effectiveness of the proposed program behavior reconstruction loss \mathcal{L}^R , which can bring the programs with similar behaviors closer in the embedding space.

Summary. The visualizations of the program embedding spaces learned by LEAPS and its ablations qualitatively justify the effectiveness of the proposed learning objectives, as complementary to the quantitative results presented in the main paper.

3.7.2 Cross Entropy Method Trajectory Visualization

As described in the main paper, once the program embedding space is learned by LEAPS, our goal becomes searching for a latent program that maximizes the reward described by a given task MDP. To this end, we adapt the Cross Entropy Method (CEM) [253], a gradient-free continuous search algorithm, to iteratively search over the program embedding space. Specifically, we iteratively perform the following steps:

1. Sample a distribution of candidate latent programs.
2. Decode the sampled latent programs into programs using the learned program decoder p_θ .
3. Execute the programs in the task environment and obtain the corresponding rewards.
4. Update the CEM sampling distribution based on the rewards.

This process is repeated until either convergence or the maximum number of sampling steps has been reached.

We perform dimensionality reduction with PCA [130] to embed the following data to a 2D space; the visualizations of CEM trajectories are shown in Figure 3.5 and Figure 3.6:

- Latent programs from the training dataset encoded by a learned encoder q_ϕ , visualized as blue scatters. There are 35k training programs. This is to visualize the shape of the program distribution in the learned program embedding space. This is also visualized in Figure 3.4.
- Ground-truth (GT) programs that exhibit optimal behaviors for solving the Karel tasks, visualized as red stars (\star). Ideally, the CEM population should iteratively move toward where the GT programs are located.
- CEM population is a batch of sampled candidate latent programs at each iteration, visualized as red scatters. Each candidate latent program can be decoded as a program that can be executed in the

task environment to obtain a reward. By averaging the reward obtained by every candidate latent program, we can calculate the average reward of this population and show it in the figures as Avg. Reward.

- CEM Next Center, visualized as cross signs (\times), indicates the center vector around which the next batch of candidate latent programs will be sampled. This vector is calculated based on a set of candidate latent programs that achieve best reward (*i.e.* elite samples) at each iteration. In this case, it is a weighted average based on the reward each candidate gets from its execution.

From Figure 3.5, we observe that both the average reward of the entire population and the reward of the next candidate program (CEM Next Center) consistently increase as the number of iterations increases, justifying the effectiveness of CEM. Moreover, we observe that the CEM population gradually moves toward where the ground-truth program is located, which aligns well with the fact that our proposed framework can reliably synthesize task-solving programs.

Yet, the populations might not always exactly converge to where the ground-truth latent program is. We hypothesize this could be attributed to the following reasons:

1. CEM convergence: while the CEM search converges, it can still be suboptimal. Since the search terminates when the next candidate latent program obtains the maximum reward (1.1 as shown in the figure) for 10 iterations, it might not exactly converge to where a ground-truth program is.
2. Dimensionality reduction: we visualized the trajectories and programs by performing dimensionality reduction from 256 to 2 dimensions with PCA, which could cause visual distortions.
3. Suboptimal learned program embedding space: while we aim to learn a program embedding space where all the programs inducing the same behaviors are mapped to the same spot in the embedding space, it is still possible that programs that induce the desired behavior can distribute to more than one

Table 3.5: Decoded linear interpolations of programs close to each other in the latent space.

Latent Program	Decoded Program
START	<code>DEF run m(turnRight move WHILE c(frontIsClear c) w(move w) WHILE c(not c(frontIsClear c) c) w(move w) IF c(frontIsClear c) i(move i) m)</code>
1	<code>DEF run m(turnRight move WHILE c(frontIsClear c) w(move w) WHILE c(not c(frontIsClear c) c) w(move w) IF c(frontIsClear c) i(move i) m)</code>
2	<code>DEF run m(turnRight move WHILE c(frontIsClear c) w(move w) IF c(not c(frontIsClear c) c) i(move i) m)</code>
3	<code>DEF run m(turnRight move WHILE c(frontIsClear c) w(move w) IF c(not c(frontIsClear c) c) i(move i) m)</code>
4	<code>DEF run m(turnRight move WHILE c(frontIsClear c) w(move w) IF c(not c(frontIsClear c) c) i(move i) m)</code>
5	<code>DEF run m(turnRight move WHILE c(frontIsClear c) w(move w) IF c(not c(frontIsClear c) c) i(move i) m)</code>
6	<code>DEF run m(turnRight move WHILE c(frontIsClear c) w(move w) IF c(not c(frontIsClear c) c) i(move i) m)</code>
7	<code>DEF run m(turnRight move turnLeft WHILE c(frontIsClear c) w(move w) IF c(not c(frontIsClear c) c) i(putMarker i) m)</code>
8	<code>DEF run m(turnRight move turnLeft WHILE c(frontIsClear c) w(move w) IF c(not c(frontIsClear c) c) i(putMarker i) m)</code>
END	<code>DEF run m(turnRight move turnLeft WHILE c(frontIsClear c) w(move w) IF c(not c(frontIsClear c) c) i(putMarker i) m)</code>

location in a learned program embedding space. Therefore, CEM search can converge to somewhere that is different from the ground-truth latent program.

On the other hand, the CEM trajectory shown in Figure 3.6 does not converge and terminates when reaching the maximum number of iterations. The ground-truth program lies far away from the initial sampled distribution, which might contribute to the difficulty of converging. This aligns with the relatively unsatisfactory performance achieved by LEAPS. Employing a more sophisticated searching algorithm or conducting a more thorough hyperparameter search could potentially improve the performance but it is not the main focus of this work.

3.7.3 Program Embedding Space Interpolations

To learn a program embedding space that allows for smooth interpolation, we propose three sources of supervision. We aim to verify the effectiveness of it by investigating interpolations in the learned program

Table 3.6: Decoded linear interpolations of programs far from each other in the latent space.

Latent Program	Decoded Program
START	<code>DEF run m(turnRight turnLeft turnLeft move turnRight putMarker move m)</code>
1	<code>DEF run m(turnRight turnLeft turnLeft move turnRight putMarker move m)</code>
2	<code>DEF run m(turnRight turnLeft turnLeft move WHILE c(frontIsClear c) w(putMarker w) turnRight move m)</code>
3	<code>DEF run m(turnRight turnLeft move turnLeft WHILE c(frontIsClear c) w(putMarker w) move m)</code>
4	<code>DEF run m(turnRight turnLeft move WHILE c(frontIsClear c) w(turnLeft w) IF c(not c(frontIsClear c) c) i(move i) m)</code>
5	<code>DEF run m(turnRight move turnLeft WHILE c(frontIsClear c) w(move w) IF c(not c(frontIsClear c) c) i(putMarker i) m)</code>
6	<code>DEF run m(move turnRight turnLeft move WHILE c(frontIsClear c) w(IF c(not c(rightIsClear c) c) i(putMarker i) w) m)</code>
7	<code>DEF run m(move turnRight turnLeft move WHILE c(frontIsClear c) w(IF c(not c(rightIsClear c) c) i(turnLeft i) w) m)</code>
8	<code>DEF run m(move turnRight move WHILE c(frontIsClear c) w(IF c(not c(rightIsClear c) c) i(turnLeft i) w) m)</code>
END	<code>DEF run m(move turnRight move WHILE c(frontIsClear c) w(IF c(not c(rightIsClear c) c) i(turnLeft i) w) m)</code>

embedding space. To this end, we follow the procedure described below to produce results shown in Table 3.5 and Table 3.6.

1. Sampling a pair of programs from the dataset (START program and END program).
2. Encoding the two programs into the learned program embedding space.
3. Linearly interpolating between the two latent programs to obtain a number of (eight) interpolated latent programs.
4. Decoding the latent programs to obtain interpolated programs (program 1 to program 8).

We show two pairs of programs and their interpolations in between below as examples. Specifically, the first pair of programs, shown in Table 3.5, are closer to each other in the latent space and the second pair of programs, shown in Table 3.6, are further from each other. We observe that the interpolations between the closer program pair exhibit smoother transitions and the interpolations between the further program pair display more dramatic change.

3.7.4 Program Evolution

In this section, we aim to investigate how predicted programs evolve over the course of searching. We visualize converged CEM search trajectories and the reward each program gets on the StairClimber task in Appendix Figure 3.5. In Table 3.7, we present the predicted programs corresponding to the CEM search trajectory on the STAIRCLIMBER task in Figure 3.5. We observe that the sampled programs consistently improve as the number of iterations increases, justifying the effectiveness of the learned program embedding and the CEM search.

3.7.5 Interpretability: Human Debugging of LEAPS Programs

Interpretability in Machine Learning is crucial for several reasons [172, 271]. First, *trust* – interpretable machine learning methods and models may more easily be trusted since humans tend to be reluctant to trust systems that they do not understand. Second, interpretability can improve the *safety* of machine learning systems. A machine learning system that is interpretable allows for diagnosing issues (e.g. the distribution shift from training data to testing data) earlier and provides more opportunities to intervene. This is especially important for safety-critical tasks such as medical diagnosis [22, 270, 92, 40, 279] and real-world robotics [39, 98, 15, 105, 333, 347, 159] tasks. Finally, interpretability can lead to *contestability*, by producing a chain of reasoning, providing insights on how a decision is made and therefore allowing humans to contest unfair or improper decisions.

We believe interpretability is especially crucial when it comes to learning a policy that interacts with the environment. In this work, we propose a framework that offers an effective way to acquire an interpretable programmatic policy structured in a program. In the following, we discuss how the proposed framework enjoys interpretability from the three aforementioned aspects. Programs synthesized by the proposed framework can naturally be better *trusted* since one can simply read and understand them. Also, through the program execution trace produced by executing a program, each decision made by the policy (*i.e.* the

Table 3.7: How predicted programs evolve throughout the course of CEM search for STAIRCLIMBER. See Figure 3.5 for the corresponding visualization of this CEM search.

Search Iteration	Best Predicted Program
Iteration: 1	<code>DEF run m(IF c(frontIsClear c) i(pickMarker i) WHILE c(leftIsClear c) w(move w) IFELSE c(frontIsClear c) i(turnRight move i) ELSE e(move e) m)</code>
Iteration: 2	<code>DEF run m(WHILE c(markersPresent c) w(move w) IFELSE c(frontIsClear c) i(turnLeft i) ELSE e(move e) WHILE c(leftIsClear c) w(move w) m)</code>
Iteration: 3	<code>DEF run m(WHILE c(not c(frontIsClear c) c) w(move turnRight w) WHILE c(leftIsClear c) w(turnLeft move w) m)</code>
Iteration: 4	<code>DEF run m(WHILE c(not c(frontIsClear c) c) w(pickMarker move w) WHILE c(leftIsClear c) w(turnLeft move w) m)</code>
Iteration: 5	<code>DEF run m(WHILE c(not c(frontIsClear c) c) w(pickMarker turnRight w) WHILE c(leftIsClear c) w(move turnLeft w) m)</code>
Iteration: 6	<code>DEF run m(WHILE c(not c(frontIsClear c) c) w(pickMarker turnRight w) WHILE c(leftIsClear c) w(move turnLeft w) m)</code>
Iteration: 7	<code>DEF run m(WHILE c(not c(leftIsClear c) c) w(turnRight w) IFELSE c(frontIsClear c) i(move i) ELSE e(turnLeft e) WHILE c(rightIsClear c) w(move w) m)</code>
Iteration: 8	<code>DEF run m(WHILE c(not c(leftIsClear c) c) w(turnRight move w) WHILE c(markersPresent c) w(turnLeft move w) m)</code>
Iteration: 9	<code>DEF run m(WHILE c(not c(noMarkersPresent c) c) w(turnRight move w) WHILE c(not c(frontIsClear c) c) w(turnLeft move w) m)</code>
Iteration: 10	<code>DEF run m(WHILE c(not c(noMarkersPresent c) c) w(turnRight move w) WHILE c(leftIsClear c) w(turnLeft move w) m)</code>
Iteration: 11	<code>DEF run m(WHILE c(not c(leftIsClear c) c) w(turnRight move w) WHILE c(noMarkersPresent c) w(turnLeft move w) m)</code>
Iteration: 12	<code>DEF run m(WHILE c(not c(leftIsClear c) c) w(turnRight move w) WHILE c(noMarkersPresent c) w(turnLeft move w) m)</code>
Iteration: 13	<code>DEF run m(WHILE c(not c(leftIsClear c) c) w(turnRight move w) WHILE c(noMarkersPresent c) w(turnLeft move w) m)</code>
Iteration: 14	<code>DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)</code>
Iteration: 15	<code>DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)</code>
Iteration: 16	<code>DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)</code>
Iteration: 17	<code>DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)</code>
Iteration: 18	<code>DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)</code>
Iteration: 19	<code>DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)</code>
Iteration: 20	<code>DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)</code>
Iteration: 21	<code>DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)</code>
Iteration: 22	<code>DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(move turnLeft w) m)</code>
Converged	<code>DEF run m(WHILE c(not c(markersPresent c) c) w(turnRight move w) WHILE c(rightIsClear c) w(turnLeft move w) m)</code>

Table 3.8: Mean return (standard deviation) [% increase in performance] after debugging by non-expert humans of LEAPS synthesized programs for 3 statement edits and 5 statement edits. Chosen LEAPS programs are median-reward programs out of 5 LEAPS seeds for each task.

Karel Task	Original Program	3 Edits	5 Edits
TOPOFF	0.86	0.95 (0.07) [10.5%]	1.0 (0.00) [16.3%]
FOURCORNER	0.25	0.75 (0.35) [200%]	0.92 (0.12) (268%)
HARVESTER	0.47	0.85 (0.05) [80.9%]	0.89 (0.00) [89.4%]
Average % Increase	-	97.1%	125%

program) is traceable and therefore satisfies the *contestability* property. Finally, the programs produced by our framework satisfy the *safety* property of interpretability as humans can diagnose and correct for issues by reading and editing the programs.

Our synthesized programs are not only readable to human users but also interactable, allowing *non-expert* users with a basic understanding of programming to diagnose and make edits to improve their performance. To test this hypothesis, we asked people with programming experience who are unfamiliar with our DSL or Karel tasks to edit suboptimal LEAPS programs to improve performance as much as possible on 3 Karel tasks: TOPOFF, FOURCORNER, and HARVESTER through a user interface displayed in Figure 3.7. Each person was given 1.5 hours (30 minutes per program), including time required to understand what the LEAPS programs were doing, understand the DSL tokens, and fully debug/test their edited programs. For each program, participants were required to modify up to 5 statements, then attempt the task again with up to only 3 modifications as calculated by the Levenshtein distance metric [162]. A single statement modification is defined as any modification/removal/addition of a IF, WHILE, IFELSE, REPEAT, or ELSE statement, or a removal/addition/change of an action statement (*e.g.* move, turnLeft, etc.). Participants were allowed to ask clarification questions, but we would not answer questions regarding how to specifically improve the performance of their program.

We display example edited programs in Figure 3.8, and the aggregated results of editing in Table 3.8. We see a significant increase in performance in all three tasks, with an average 97.1% increase in performance with 3 edits and an average 125% increase in performance with 5. These numbers are averaged over 3 people,

with standard deviations reported in the table. Thus we see that even slight modifications to suboptimal LEAPS programs can enable much better Karel task performance when edited by non-expert humans.

Our experiments in this section make an interesting connection to works in program/code repair (*i.e.* automatic bug fixing) [340, 211, 129, 330, 93, 266, 146, 72, 166, 43, 323, 103, 319, 192], where the aim is to develop algorithms and models that can find bugs or even repair programs without the intervention of a human programmer. While the goal of these works is to fix programs produced by humans, our goal in this section is to allow humans to improve programs synthesized by the proposed framework.

Another important benefit of programmatic policies is *verifiability* - the ability to verify different properties of policies such as correctness, stability, smoothness, robustness, safety, etc. Since programmatic policies are highly structured, they are more amenable to formal verification methods developed for traditional software systems as compared to neural policies. Recent works [25, 307, 306, 351] show that various properties of programmatic policies (programs written using DSLs, decision trees) can be verified using existing verification algorithms, which can also be applied to programs synthesized by the proposed framework.

3.7.6 Optimal and Synthesized Programs

In this section, we present the programs from the testing set which are selected for conducting ablation studies in the main paper in Figure 3.11. Also, we manually write programs that induce optimal behaviors to solve the Karel tasks and present them in Figure 3.11. Note that while we only show one optimal program for each task, there exist multiple programs that exhibit the desired behaviors for each task. Then, we analyze the program reconstructed by LEAPS, its ablations, and the naïve program synthesis baseline in Section 3.7.6.1, and discuss the programs synthesized by LEAPS for Karel tasks in Section 3.7.6.2.

3.7.6.1 Program Behavior Reconstruction

This section serves as a complement to the ablation studies in the main paper, where we aim to justify the effectiveness of the proposed framework and the learning objectives. To this end, we select programs that are unseen to LEAPS and its ablations during the learning program embedding space from the testing set and reconstruct those programs using LEAPS, its ablations and the naïve program synthesis baseline. Those selected programs are shown in Figure 3.11 and the reconstructed programs are shown in Figure 3.12.

The naïve program synthesis baseline fails on the complex WHILE+2IF+IFELSE program, as it rarely synthesizes conditional and loop statements, instead generating long sequences of action tokens that attempt to replicate the desired behavior of those statements. We believe that this is because it is incentivized to initially predict action tokens to gain more immediate reward, making it less likely to synthesize other tokens. LEAPS and its variations perform better and synthesize more complex programs, demonstrating the importance of the proposed two-stage learning scheme in biasing program search. Also, LEAPS synthesizes programs that are more concise and induce behaviors which are more similar to given testing programs, justifying the effectiveness of the proposed learning objectives.

3.7.6.2 Karel Environment Tasks

This section is complementary to the main experiments in the main paper, where we compare LEAPS against the baselines on a set of Karel tasks, which is described in detail in Section 3.7.11. The programs synthesized by LEAPS are presented in Figure 3.14.

The synthesized programs solve both STAIRCLIMBER and MAZE. For TOPOFF, since the average expected number of markers presented in the last row is 3, LEAPS synthesizes a sub-optimal program that conducts the topoff behavior three times. For CLEANHOUSE, while all the baselines fail on this task, the synthesized program achieves some performance by simply moving around and try to pick up markers. For HARVESTER,

Table 3.9: Extended reward comparison on original tasks with 8×8 or 12×12 grids and zero-shot generalization to 100×100 grids. LEAPS achieves the best generalization performance on all the tasks except for HARVESTER.

		STAIRCLIMBER	MAZE	FOURCORNER	TOPOFF	HARVESTER
DRL	Original	1.00 (0.00)	1.00 (0.00)	0.29 (0.05)	0.32 (0.07)	0.90 (0.10)
	100x100	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.01 (0.01)	0.00 (0.00)
DRL-abs	Original	0.13 (0.29)	1.00 (0.00)	0.36 (0.44)	0.63 (0.23)	0.32 (0.18)
	100x100	0.00 (0.00)	0.04 (0.05)	0.37 (0.44)	0.15 (0.12)	0.02 (0.01)
DRL-FCN	Original	1.00 (0.00)	0.97 (0.03)	0.20 (0.34)	0.28 (0.12)	0.46 (0.16)
	100x100	-0.20 (0.10)	0.01 (0.01)	0.00 (0.00)	0.01 (0.01)	0.02 (0.00)
VIPER	Original	0.02 (0.02)	0.69 (0.05)	0.40 (0.42)	0.30 (0.06)	0.51 (0.07)
	100x100	0.00 (0.00)	0.10 (0.12)	0.40 (0.42)	0.03 (0.00)	0.04 (0.00)
LEAPS	Original	1.00 (0.00)	1.00 (0.00)	0.45 (0.40)	0.81 (0.07)	0.45 (0.28)
	100x100	1.00 (0.00)	1.00 (0.00)	0.45 (0.37)	0.21 (0.03)	0.00 (0.00)

LEAPS fails to acquire the desired behavior that required nested loops but produces a sub-optimal program that contains only action tokens.

3.7.7 Additional Generalization Experiments

Here, we present additional generalization experiments to complement those presented in Section 3.5.6. In Section 3.7.7.1, we extend the 100x100 state size zero-shot generalization experiments to 3 additional tasks. In Section 3.7.7.2, we analyze how well baseline methods and LEAPS can generalize to unseen configurations of a given task.

3.7.7.1 Generalization on FOURCORNER, TOPOFF, and HARVESTER

Evaluating zero-shot generalization performance assumes methods to work reasonably well on the original tasks. For this reason (and due to space limitations) we present only STAIRCLIMBER and MAZE for generalization experiments in the main text in Section 3.5.6 because most methods achieve reasonable performance on these two tasks, with DRL and LEAPS both solving these tasks fully and DRL-abs solving Maze fully.

However, here we also present full results for all tasks except CLEANHOUSE (as no method except LEAPS has a reasonable level of performance on it). The results are summarized in Table 3.9. We see that LEAPS generalizes well on FOURCORNER and maintains the best performance on TOPOFF. It is outperformed

on HARVESTER, although none of the methods do well on HARVESTER as the highest obtained reward by any method is 0.04 (by VIPER). In summary, LEAPS performs the best on 4 out of these 5 tasks, further demonstrating its superior zero-shot generalization performance.

Furthermore, we note that it is possible that a DRL policy employing a fully convolutional network (FCN) as proposed in Long, Shelhamer, and Darrell [177] can handle varying observation sizes. FCNs were also demonstrated in Silver et al. [278] to demonstrate better generalization performance than traditional convolutional neural network policies. However, we hypothesize that the generalization performance here will still be poor as there is a large increase in the number of features that the FCN architecture needs to aggregate when transferring from 8x8/12x12 state inputs to 100x100 inputs—a 10x input size increase that FCN is not specifically designed to deal with. We have included both FCN’s zero-shot generalization results and its results on the original grid sizes in Table 3.9. DRL-FCN, where we have replaced the policy and value function networks of PPO with an FCN, does manage to perform zero-shot transfer marginally better than DRL performs when training from scratch (as it DRL’s architecture cannot handle varied input sizes) on MAZE and HARVESTER. However, it obtains a negative reward on STAIRCLIMBER as it attempts to navigate away from the stairs when transferring to the 100×100 grid size. Its performance is still far worse than LEAPS and VIPER on most tasks, demonstrating that the programmatic structure of the policy is important for these tasks.

3.7.7.2 Generalization to Unseen Configurations

We present a generalization experiment in the main paper to study how well the baselines and the programs synthesized by the proposed framework can generalize to larger state spaces that are unseen during training without further learning on the STAIRCLIMBER and MAZE tasks. In this section, we investigate the ability of generalizing to different configurations, which are defined based on the marker placement related to solve a task, on both the TopOFF task and HARVESTER task.

Table 3.10: Mean return (standard deviation) [% change in performance] on generalizing to unseen configurations on TopOFF and HARVESTER task.

TopOFF		Training configuration %				
		75%	50%	25%	10%	5%
DRL	0.17 (0.05) [-46.8%]	0.12 (0.09) [-62.5%]	0.12 (0.06) [-62.5%]	0.17 (0.13) [-46.8%]	0.13 (0.04) [-59.4%]	
DRL-abs	0.23 (0.29) [-63.5%]	0.29 (0.36) [-54.0%]	0.45 (0.45) [-28.6%]	0.24 (0.38) [-61.9%]	0.26 (0.37) [-18.8%]	
VIPER	0.27 (0.03) [-10.0%]	0.28 (0.04) [-6.67%]	0.27 (0.06) [-10.0%]	0.27 (0.02) [-10.0%]	0.28 (0.03) [-6.67%]	
LEAPS	0.68 (0.18) [-15.0%]	0.65 (0.13) [-18.8%]	0.61 (0.24) [-23.8%]	0.68 (0.21) [-15.0%]	0.67 (0.18) [-16.3%]	

HARVESTER		Training configuration %				
		75%	50%	25%	10%	5%
DRL	0.64 (0.24) [-28.9%]	0.71 (0.29) [-21.1%]	0.21 (0.06) [-76.7%]	0.14 (0.09) [-84.4%]	0.04 (0.01) [-95.6%]	
DRL-abs	0.14 (0.21) [-56.3%]	0.24 (0.25) [-25.0%]	0.05 (0.06) [-84.4%]	0.13 (0.21) [-59.4%]	0.31 (0.31) [-3.13%]	
VIPER	0.54 (0.01) [+5.88%]	0.54 (0.02) [+5.88%]	0.55 (0.01) [+7.84%]	0.54 (0.01) [+5.88%]	0.44 (0.22) [-13.7%]	
LEAPS	0.40 (0.30) [-13.0%]	0.42 (0.27) [-8.69%]	0.50 (0.35) [+08.69%]	0.12 (0.19) [-73.9%]	0.01 (0.03) [-97.6%]	

Since solving TopOFF requires an agent to put markers on top of all markers on the last row, the initial configurations are determined by the marker presence on the last row. The grid has a size of 10×10 inside the surrounding wall. We do not spawn a marker at the bottom right corner in the last row, leaving 9 possible locations with marker, allowing 2^9 possible initial configurations. On the other hand, HARVESTER requires an agent to pick up all the markers placed in the grid. The grid has a size of 6×6 inside the surrounding wall, leaving 36 possible locations in grid with a marker, resulting in 2^{36} possible initial configurations.

We aim to test if methods can learn from only a small portion of configurations during training and still generalize to all the possible configurations without further learning. To this end, we experiment using 75%, 50%, 25%, 10%, 5% of the configurations for training DRL, DRL-abs, and VIPER and for the program search stage of LEAPS. Then, we test zero-shot generalization of the learned models and programs on all the possible configurations. We report the performance in Table 3.10. We compare the performance each method achieves to its own performance learning from all the configurations (reported in the main paper) to investigate how limiting training configurations affects the performance. Note that the results of training and testing on 100% configurations are reported in the main paper, where no generalization is required.

TOPOFF. LEAPS outperforms all the baselines on the mean return on all the experiments. VIPER and LEAPS enjoy the lowest and the second lowest performance decrease when learning from only a portion of configurations, which demonstrates the strength of programmatic policies. DRL-abs slightly outperforms DRL, with better absolute performance and lower performance decrease. We believe that this is because DRL takes entire Karel grids as input, and therefore held out configurations are completely unseen to it. In contrast, DRL-abs takes abstract states (*i.e.* local perceptions) as input, which can alleviate this issue.

HARVESTER. VIPER outperforms almost all other methods on absolute performance and performance decrease, while LEAPS achieves second best results, which again justifies the generalization of programmatic policies. Both DRL and DRL-abs are unable to generalize well when learning from a limited set of configurations, except in the case of DRL-abs learning from 5% of configurations, which can be attributed to the high-variance of DRL-abs results.

3.7.8 Additional Analysis on Experimental Results

Due to the limited space in the main paper, we include additional analysis of the experimental results in this section.

3.7.8.1 DRL vs. DRL-abs

We hypothesize that DRL-abs does not always outperform DRL due to imperfect perception (*i.e.* state abstraction) design. DRL-abs takes abstract states as input (*i.e.* `frontIsClear()`, `leftIsClear()`, `rightIsClear()`, `markerPresent()` in our design), which only describe local perception while omitting the information of the entire map. Therefore, for tasks such as STAIRCLIMBER, HARVESTER, and CLEANHOUSE, which would be easier to solve with access to the entire Karel grid, DRL might outperform DRL-abs. In this work, DRL-abs' abstract states are the perceptions from the DSL we synthesize programs with to make the comparisons fair against our method as well as analyzing the effects of abstract states in the DRL domain. However, a

more sophisticated design for perception/state abstraction could potentially improve the performance of DRL-abs.

3.7.8.2 VIPER Generalization

VIPER operates on the abstract state space which is invariant to grid size. However, for the reasons below, it is still unable to transfer the behavior to the larger grid despite its abstract state representation. We hypothesize that VIPER’s performance suffers on zero-shot generalization for two main reasons.

1. It is constrained to imitate the DRL teacher policy during training, which is trained on the smaller grid sizes. Thus its learned policy also experiences difficulty in zero-shot generalization to larger grid sizes.
2. Its decision tree policies cannot represent certain looping behaviors as they simply perform a one-to-one mapping from abstract state to action, thus making it difficult to learn optimal behaviors that require a one-to-many mapping between an abstract state and a set of desired actions. Empirically, we observed that training losses for VIPER decision trees were much higher for tasks such as STAIRCLIMBER which require such behaviors.

3.7.9 Detailed Descriptions and Illustrations of Ablations and Baselines

This section provides details on the variations of LEAPS used for ablations studies and the baselines which we compare against. The descriptions of the ablations of LEAPS are presented in Section 3.7.9.1 and the illustrations are shown in Figure 3.15. The naïve program synthesis baseline is illustrated in Figure 3.16 (c) for better visualization. Then, the descriptions of the baselines are presented in Section 3.7.9.2 and the illustrations are shown in Figure 3.16.

3.7.9.1 Ablations

We first ablate various components of our proposed framework in order to (1) justify the necessity of the proposed two-stage learning scheme and (2) identify the effects of the proposed objectives. We consider the following baselines and ablations of our method.

- Naïve: the naïve program synthesis baseline is a policy that learns to directly synthesize a program from scratch by recurrently predicting a sequence of program tokens. The architecture of this baseline is a recurrent neural network which takes an initial starting token as the input at the first time step, and then sequentially outputs a program token at each time step to compose a program until an end token is produced. Note that the observation of this baseline is its own previously outputted program token instead of the state of the task environment (e.g. Karel grids). Also, at each time step, this baseline produces a distribution over all the possible program tokens in the given DSL instead of a distribution over agent’s action in the task environment (e.g. `move()`). This baseline investigates if an end-to-end learning method can solve the problem. This baseline is illustrated in Figure 3.16 (c).
- LEAPS-P: the simplest ablation of LEAPS, in which the program embedding space is learned by only optimizing the program reconstruction loss \mathcal{L}^P . This baseline is illustrated in Figure 3.15 (a).
- LEAPS-P+R: an ablation of LEAPS which optimizes both the program reconstruction loss \mathcal{L}^P and the program behavior reconstruction loss \mathcal{L}^R . This baseline is illustrated in Figure 3.15 (b).
- LEAPS-P+L: an ablation of LEAPS which optimizes both the program reconstruction loss \mathcal{L}^P and the latent behavior reconstruction loss \mathcal{L}^L . This baseline is illustrated in Figure 3.15 (c).
- LEAPS (LEAPS-P+R+L): LEAPS with all the losses, optimizing our full objective.
- LEAPS-rand-{8/64}: like LEAPS, this ablation also optimizes the full objective for learning the program embedding space. But when searching latent programs, instead of CEM, it simply randomly samples

8/64 candidate latent programs and chooses the best performing one. These baselines justify the effectiveness of using CEM for searching latent programs.

3.7.9.2 Baselines

We evaluate LEAPS against the following baselines (illustrated in Figure 3.16).

- DRL: a neural network policy trained on each task and taking raw states (Karel grids) as input. A Karel grid is represented as a binary tensor with dimension $W \times H \times 16$ (there are 16 possible states for each grid square) instead of an image. This baseline is illustrated in Figure 3.16 (a).
- DRL-abs: a recurrent neural network policy directly trained on each Karel task but instead of taking raw states (Karel grids) as input it takes *abstract* states as input (*i.e.* it sees the same perceptions as LEAPS). Specifically, all returned values of perceptions such as `frontIsClear() == true`, `leftIsClear() == false`, `rightIsClear() == true`, `markersPresent() == false`, and `noMarkersPresent() == true` are concatenated as a binary vector, which is then fed to the DRL-abs policy as its input. This baseline allows for a fair comparison to LEAPS since the program execution process also utilizes abstract state information. This baseline is illustrated in Figure 3.16 (b).
- DRL-abs-t: a DRL *transfer* learning baseline in which for each task, we train DRL-abs policies on all other tasks, then fine-tune them on the current task. Thus it acquires a prior by learning to first solve other Karel tasks. Rewards are reported for the policies from the task that transferred with highest return. We only transfer DRL-abs policies as some tasks have different state spaces so that transferring a DRL policy trained on a task to another task with a different state space is not possible. This baseline is designed to investigate if acquiring task related priors allows DRL policies to perform better on our Karel tasks. Unlike LEAPS, which acquires priors from a dataset consisting of randomly

generated programs and the behaviors those program induce in the environment, DRL-abs-t allows for acquiring priors from goal-oriented behaviors (*i.e.* other Karel tasks).

- HRL: a hierarchical RL baseline in which a VAE is first trained on action sequences from program execution traces used by LEAPS. Once trained, the decoder is utilized as a low-level policy for learning a high-level policy to sample actions from. Similar to LEAPS, this baseline utilizes the dataset to produce a prior of the domain. It takes raw states (Karel grids) as input.

This baseline is also designed to investigate if acquiring priors allow DRL policies to perform better. Similar to LEAPS, which acquires priors from a dataset consisting of randomly generated programs and the behaviors those program induce in the environment, HRL is trained to acquire priors by learning to reconstruct the behaviors induced by the programs. One can also view this baseline as a version of the framework proposed in [108] with some simplifications, which also learns an embedding space using a VAE and then trains a high-level policy to utilize this embedding space together with the low-level policy whose parameters are frozen. This baseline is illustrated in Figure 3.16 (d).

- HRL-abs: the same method as HRL but taking abstract states (*i.e.* local perceptions) as input. This baseline is illustrated in Figure 3.16 (d).
- VIPER [25]: A decision-tree programmatic policy which imitates the behavior of a deep RL teacher policy via a modified DAgger algorithm [251]. This decision tree policy cannot synthesize loops, allowing us to highlight the performance advantages of more expressive program representation that LEAPS is able to take advantage of.

All the baselines are trained with PPO [265] or SAC [104], including the VIPER teacher policy. More training details can be found in Section 3.7.12.

3.7.10 Program Dataset Generation Details

To learn a program embedding space for the proposed framework and its ablations, we randomly generate 50k programs to form a dataset with 35k training programs and 7.5k programs for validation and testing. Simply generating programs by uniformly sampling all the tokens from the DSL would yield programs that mainly only contain action tokens since the chance to synthesize conditional statements with correct grammar is low. Therefore, to produce programs that are longer and deeply nested with conditional statements to induce more complex behaviors, we propose to sample programs using a probabilistic sampler.

To generate each program, we sample program tokens according to the probabilities listed in Table 3.11 at every step until we sample an ending token or when a maximum program length is reached. When generating programs, we ensure that no program is identical to any other. Each token is generated sequentially, and length is effectively governed by the STMT_STMT token detailed in Table 3.11’s caption. There is a maximum depth limit of 4 nested conditional/loop statements, and a maximum statement depth limit of 6 (can’t have more than 6 nested STMT_STMT tokens). Note that this sampling procedure does not guarantee that the programs generated will terminate, hence when executing them to obtain ground-truth interactions for training the Program Behavior and Latent Behavior Reconstruction losses we limit the max program execution length to 100 environment timesteps. This sampling procedure results in the distribution of program lengths seen in Figure 3.17.

Intuitively, shorter lengths can bias synthesized programs to compress the same behaviors into fewer tokens through the use of loops, making program search easier. Therefore, in our experiments, we have limited the maximum output program length of LEAPS to 45 tokens (as the maximum in the dataset is 44). As shown in the example programs generated by LEAPS in Figure 3.14, LEAPS successfully generates loops for our Karel tasks, which can be probably attributed to this bias of program length. We further verify this intuition by rerunning LEAPS with the max program length set to 100 tokens on the Karel tasks. We

display generated programs in Table 3.12, where we see that some of the generated programs are indeed much longer and lack loop statements and structures.

Table 3.11: The probability of sampling program tokens when generating the program dataset. Tokens are generated sequentially, and STMT_STMT refers to breaking up the current token into two tokens, each of which is selected according to the same probability distribution again. Thus it effectively controls how long programs will be.

	WHILE	REPEAT	STMT_STMT	ACTION	IF	IFELSE
Standard Dataset	0.15	0.03	0.5	0.2	0.08	0.04

3.7.11 Karel Task Details

MDP Tasks We utilize environment state based reward functions for the RL tasks STAIRCLIMBER, FOUR-CORNER, TOPOFF, MAZE, HARVESTER, and CLEANHOUSE. For each task, we average performance of the policies on 10 random environment start configurations. For all tasks with marker placing objectives, the final reward will be 0—regardless of the any other agent actions—if a marker is placed in the wrong location. This is done in order to discourage “spamming” marker placement on every grid location to exploit the reward functions. All rewards described below are then normalized so that the return is between [0, 1.0] for tasks without penalties, and [-1.0, 1.0] for tasks with negative penalties, for easier learning for the DRL methods. We visualize all tasks as well as their start and ideal end states in Figure 3.18 on a 10×10 grid for consistency in the visualizations (except CLEANHOUSE).

3.7.11.1 STAIRCLIMBER

The goal is to climb the stairs to reach where the marker is located. The reward is defined as a sparse reward: 1 if the agent reaches the goal in the environment rollout, -1 if the agent moves to a position off of the stairs during the rollout, and 0 otherwise. This is on a 12×12 grid, and the marker location and agent’s initial location are randomized between rollouts.

Table 3.12: LEAPS Length 100 Synthesized Karel Programs. Line breaks are not shown here as the programs are very long. The examples picked are ones that represent the programs generated by most seeds for each task. Without the 45 token restriction on program lengths, programs for TOPOFF, FOURCORNER, and HARVESTER are very long and have repetitive movements that can easily be put into REPEAT or WHILE loops. The CLEANHOUSE program also contains repeated, somewhat redundant WHILE loops. MAZE and STAIRCLIMBER programs are mostly unaffected by the change in maximum program length. These programs demonstrate that the bias induced by program length restriction is important for producing more complex programs in the program synthesis phase of LEAPS.

3.7.11.2 FOURCORNER

The goal is to place a marker at each corner of the Karel environment grid. The reward is defined as sum of corners having a marker divided by four. If the Karel state has a marker placed in wrong location, the reward will be 0. This is on a 12×12 grid.

3.7.11.3 TOPOFF

The goal is to place a marker wherever there is already a marker in the last row of the environment, and end up in the rightmost square on the bottom row at the end of the rollout. The reward is defined as the number of consecutive places until the agent either forgets to place a marker where the marker is already present or places a marker at an empty location in last row, with a bonus for ending up on the last square. This is on a 12×12 grid, and the marker locations in the last row are randomized between rollouts.

3.7.11.4 MAZE

The goal is to find a marker in randomly generated maze. The reward is defined as a sparse reward: 1 if the agent finds the marker in the environment rollout, 0 otherwise. This is on a 8×8 grid, and the marker location, agent's initial location, and the maze configuration itself are randomized between rollouts.

3.7.11.5 CLEANHOUSE

We design a complex 14×22 Karel environment grid that resembles an apartment. The goal is to pick up the garbage (markers) placed at 10 different locations and reach the location where there is a dustbin (2 markers in 1 location). To make the task simpler, we place the markers adjacent to any wall in the environment. The reward is defined as total locations cleaned (markers picked) out of the total number of markers placed in initial Karel environment state (10). The agent's initial location is fixed but the marker locations are randomized between rollouts.

3.7.11.6 HARVESTER

The goal is to pickup a marker from each location in the Karel environment. The final reward is defined as the number of markers picked up divided the total markers present in the initial Karel environment state. This is on a 8×8 grid. We run both MAZE and HARVESTER on smaller Karel environment grids to save time and compute resources because these are long horizon tasks.

3.7.12 Hyperparameters and Training Details

3.7.12.1 DRL and DRL-abs

RL training directly on the Karel environment is performed with the PPO algorithm [265] for 2M timesteps using the ALF codebase*. We tried a discretized SAC [104] implementation (by replacing Gaussian distributions with Categorical distributions), but it was outperformed by PPO on the Karel tasks on all environments. We also tried tabular Q-learning from raw Karel grids (it wouldn't work well on abstract states as the state is partially observed), however it was also consistently outperformed by PPO. For DRL, the policies and value networks are the same with a shared convolutional encoder that first processes the state (as the Karel state size is $(H \times W \times 16)$ for 16 possible agent direction or marker placement values that each state in the grid can take on at a time. The convolutional encoder consists of two layers: the first with 32 filters, kernel size 2, and stride 1, the second with 32 filters, kernel size 4, and stride 1. For DRL-abs, the policy and value networks are both comprised of an LSTM layer and a 2-layer fully connected network, all with hidden sizes of 100.

For each task, we perform a comprehensive hyperparameter grid search over the following parameters, and report results from the run with the best averaged final reward over 5 seeds.

The hyperparameter grid is listed below, shared parameters are also listed:

- Importance Ratio Clipping: {0.05, 0.1, 0.2}

*<https://github.com/HorizonRobotics/alf/>

- Advantage Normalization: {True, False}
- Entropy Regularization: {0.1, 0.01, 0.001}
- Number of updates per training iteration (This controls the ratio of gradient steps to environment steps): {1, 4, 8, 16}
- Number of environment steps per set of training iterations: 32
- Number of parallel actors: 10
- Optimizer: Adam
- Learning Rate: 0.001
- Batch Size: 128

Hyperparameters that performed best for each task are listed below.

DRL	Import Ratio Clip	Adv Norm	Entropy Reg	Updates per Train Iter
CLEANHOUSE	0.1	True	0.01	4
FOURCORNER	0.2	True	0.01	16
HARVESTER	0.05	True	0.01	8
MAZE:	0.05	True	0.001	8
STAIRCLIMBER	0.1	True	0.1	4
TOPOFF	0.05	True	0.001	4

	DRL-abs	Import Ratio Clip	Adv Norm	Entropy Reg	Updates per Train Iter
CLEANHOUSE	0.2	True	0.01	8	
FOURCORNER	0.05	True	0.01	4	
HARVESTER	0.2	True	0.01	4	
MAZE:	0.2	True	0.001	4	
STAIRCLIMBER	0.05	True	0.1	16	
TOPOFF	0.2	True	0.001	8	

3.7.12.2 DRL-abs-t

DRL-abs-t is limited to DRL-abs policies as the state spaces are different for some of the Karel tasks. For DRL-abs-t, we use the best hyperparameter configuration for each Karel task to train a policy to 1M timesteps. Then, we attempt direct policy transfer to each other task by training for another 1M timesteps on the new task with the same hyperparameters (excluding transferring to the same task). Numbers reported are from the task transfer that achieved the highest reward. The tasks that we transfer from for each task are listed below:

DRL-abs-t	Transferred from
CLEANHOUSE	HARVESTER
FOURCORNER	TOPOFF
HARVESTER	MAZE
MAZE	STAIRCLIMBER
STAIRCLIMBER	HARVESTER
TOPOFF	HARVESTER

3.7.12.3 HRL

Pretraining stage: We first train a VAE to reconstruct action trajectories generated from our program dataset. For each program, we generate 10 rollouts in randomly configured Karel environments to produce the HRL dataset, giving this baseline the same data as LEAPS. These variable-length action sequences are encoded via an LSTM encoder into a 10-dimensional, continuous latent space and decoded by an LSTM decoder into the original action trajectories. We chose 10-dimensional so as to not make downstream RL too difficult. We tune the KL divergence weight (β) of this network such that it's as high as possible while being able to reconstruct the trajectories well. Network/training details below:

- β : 1.0
- Optimizer: Adam (All optimizers)
- Learning Rates: 0.0003
- Hidden layer size: 128
- # LSTM layers (both encoder/decoder): 2

- Latent embedding size: 10
- Nonlinearity: ReLU
- Batch Size: 128

Downstream (Hierarchical) RL On our Karel tasks, we use the VAE’s decoder to decode latent vectors (actions for the RL agent) into varied-length action sequences for all Karel tasks. The decoder parameters are frozen and used for all environments. The RL agent is retrained from scratch for each task, in the same manner as the standard RL baselines DRL-abs and DRL. We use Soft-Actor Critic (SAC, Haarnoja et al. [104]) as the RL algorithm as it is state of the art in many continuous action space environments. SAC grid search parameters for all environments follow below:

- Number of updates per training iteration: {1, 8}
- Number of environment steps per set of training iterations: 8 (multiplied by the number of steps taken by the decoder in the environment)
- Polyak Averaging Coefficient: {0.95, 0.9}
- Number of parallel actors: 1
- Batch size: 128
- Replay buffer size: 1M

The best hyperparameters follow:

HRL-abs	Updates per Train Iter	Polyak Coefficient
CLEANHOUSE	1	0.95
FOURCORNER	8	0.9
HARVESTER	8	0.95
MAZE	1	0.95
STAIRCLIMBER	1	0.9
TOPOFF	1	0.9

HRL	Updates per Train Iter	Polyak Coefficient
CLEANHOUSE	1	0.9
FOURCORNER	1	0.95
HARVESTER	1	0.95
MAZE	8	0.9
STAIRCLIMBER	8	0.95
TOPOFF	8	0.95

3.7.12.4 Naïve

The naïve program synthesis baseline takes an initial token as input and outputs an entire program at each timestep to learn a recurrent policy guided by the rewards of these programs. We execute these generated programs on 10 random environment start configurations in Karel to get the reward. We run PPO for 2M Karel environment timesteps. The policy network is comprised of one shared GRU layer, followed by two fully connected layers, for both the policy and value networks. For evaluation, we generate 64 programs

from the learned policy, and choose the program with the maximum reward on 10 demonstrations. For each task, we perform a hyperparameter grid search over the following parameters, and report results from the run with the best averaged final reward over 5 seeds. We exponentially decay the entropy loss coefficient in PPO from the initial to final entropy coefficient to avoid local minima during the initial training steps.

- Learning Rate: 0.0005
- Batch Size (B): {64, 128, 256}
- initial entropy coefficient (E_i): {1.0, 0.1}
- final entropy coefficient: {0.01}
- Hidden Layer Size: 64

Hyperparameters that performed best for each task are listed below.

Naïve	B	E_i
WHILE	128	0.1
IFELSE+WHILE	256	1.0
2IF+IFELSE	256	0.1
WHILE+2IF+IFELSE	128	0.1

	Naïve	B	E_i
CLEANHOUSE	128	0.1	
FOURCORNER	128	1.0	
HARVESTER	128	1.0	
MAZE	256	1.0	
STAIRCLIMBER	128	1.0	
TOPOFF	128	1.0	

3.7.12.5 VIPER

VIPER [25] builds a decision tree programmatic policy by imitating a given teacher policy. We use the best DRL policies as teachers instead of the DQN [200] teacher policy used in Bastani, Pu, and Solar-Lezama [25]. We did this in order to give the teacher the best performance possible for maximum fairness in comparison against VIPER, as we empirically found the PPO policy to perform much better on our tasks than a DQN policy.

We perform a grid search over VIPER hyperparameters, listed below:

- Max depth of decision tree: {6, 12, 15}
- Max number of samples for tree policy: {100k, 200k, 400k}
- Sample reweighting: {True, False}

The best hyperparameters found for each task are listed below:

VIPER	Max Depth	Max Num Samples	Sample Reweighting
CLEANHOUSE	6	100k	False
FOURCORNER	12	100k	False
HARVESTER	12	400k	True
MAZE	12	100k	True
STAIRCLIMBER	12	400k	True
TOPOFF	15	100k	False

3.7.12.6 Program Embedding Space VAE Model

Encoder-Decoder Architecture. The encoder and decoder are both recurrent networks. The encoder structure consists of a PyTorch token embedding layer, then a recurrent GRU cell, and two linear layers that produce μ and $\log \sigma$ vectors to sample the program embedding.

The decoder consists of a recurrent GRU cell which takes in the embedding of the previous token generated and then a linear token output layer which models the log probabilities of all discrete tokens. Since we have access to DSL grammar during program synthesis, we utilize a syntax checker based on the Karel DSL grammar from Bunel et al. [35] at the output of the decoder to limit predictions to syntactically valid tokens. We restrict our decoder from predicting syntactically invalid programs by masking out tokens that make a program syntactically invalid at each timestep. This syntax checker is designed as a state machine that keeps track of a set of valid next tokens based on the current token, open code blocks (e.g. `while`, `if`, `ifelse`) in the given partial program, and the grammar rules of our DSL. Since we

generate a program as a sequence of tokens, the syntax checker outputs at each timestep a mask M , where $M \in \{-\infty, 0\}^{\text{number of DSL tokens}}$, and

$$M_j = \begin{cases} -\infty & \text{if the } j\text{-th token is not valid in the current context} \\ 0 & \text{otherwise} \end{cases}$$

This mask is added to the output of the last layer of the decoder, just before the Softmax operation that normalizes the output to a probability over the tokens.

π Architecture. The program-embedding conditioned policy π consists of a GRU layer that operates on the inputs and three MLP layers that output the log probability of environment actions. Specifically, it takes a latent program vector, current environment state, and previous action as input and outputs the predicted environment action for each timestep.

To evaluate how close the predicted neural execution traces are to the execution traces of the ground-truth programs, we consider the following metrics:

- Action token accuracy: the percentage of matching actions in the predicted execution traces and the ground-truth execution traces.
- Action sequence accuracy: the percentage of matching action sequences in the predicted execution traces and the ground-truth execution traces. It requires that a predicted execution trace entirely matches the ground-truth execution trace.

After convergence, our model achieves an action token accuracy of 96.5% and an action sequence accuracy of 91.3%.

Training. The reinforcement learning algorithm used for the program behavior reconstruction \mathcal{L}^R is REINFORCE [324].

When training LEAPS with all losses, we first train with the Program Reconstruction (\mathcal{L}^P) and Latent Behavior Reconstruction (\mathcal{L}^L) losses, essentially setting $\lambda_1 = \lambda_3 = 1$ and $\lambda_2 = 0$ of our full objective, reproduced below:

$$\min_{\theta, \phi, \pi} \lambda_1 \mathcal{L}_{\theta, \phi}^P(\rho) + \lambda_2 \mathcal{L}_{\theta, \phi}^R(\rho) + \lambda_3 \mathcal{L}_\pi^L(\rho, \pi), \quad (3.6)$$

Once this model is trained for one epoch, we then train exclusively with the Program Behavior Reconstruction loss (\mathcal{L}^R), setting $\lambda_2 = 1$ and $\lambda_1 = \lambda_3 = 0$, with equal number of updates. These two update steps are repeated alternatively till convergence is achieved. This is done to avoid potential issues of updating with supervised and reinforcement learning gradients at the same time. We did not attempt to train these 3 losses jointly.

All other shared hyperparameters and training details are listed below:

- β : 0.1
- Optimizer: Adam (All optimizers)
- Supervised Learning Rate: 0.001
- RL Learning Rate: 0.0005
- Batch Size: 256
- Hidden Layer Size: 256
- Latent Embedding Size: 256
- Nonlinearity: $Tanh()$

3.7.12.7 Cross-Entropy Method (CEM)

CEM search works as follows: we sample an initial latent program vector from the initial distribution D_I , and generate population of latent program vectors from a $\mathcal{N}(0, \sigma I_d)$ distribution, where I_d is the identity matrix of dimension d . The samples are added to the initial latent program vector to obtain the population of latent program vectors which are decoded into programs to obtain their rewards. The population is then sorted based on rewards obtained, and a set of ‘elites’ with the highest reward are reduced using weighted mean to one latent program vector for the next iteration of sampling. This process repeats for all CEM iterations.

We include the following sets of hyperparameters when searching over the program embedding space to maximize R_{mat} to reproduce ground-truth program behavior or to maximize R_{mat} in the Karel task MDP.

- Population Size (S): {8, 16, 32, 64}
- μ : {0.0}
- σ : {0.1, 0.25, 0.5}
- % of population elites (this refers to the percent of the population considered ‘elites’): {0.05, 0.1, 0.2}
- Exponential σ decay[†]: {True, False}
- Initial distribution D_I : $\{\mathcal{N}(1, \mathbf{0}), \mathcal{N}(0, I_d), \mathcal{N}(0, 0.1I_d)\}$

Since a comprehensive grid search over the hyperparameter space would be too computationally expensive, we choose parameters heuristically. We report results from the run with the best averaged reward over 5 seeds. Hyperparameters that performed best for each task are listed below.

Ground-Truth Program Reconstruction We include the following sets of hyperparameters when searching over the program embedding space to maximize R_{mat} to reproduce ground-truth program behavior.

[†]Over the first 500 epochs, we exponentially decay σ to 0.1, and then we keep it at 0.1 for the rest of the epochs if True.

We allow the search to run for 1000 CEM iterations, counting the search as a success when it achieves 10 consecutive CEM iterations with matching the ground-truth program behaviors exactly in the environment across 10 random environment start configurations. We use same hyperparameter set to compare LEAPS-P, LEAPS-P+R, LEAPS-P+L, and LEAPS.

CEM	S	σ	# Elites	Exp Decay	D_I
WHILE	32	0.25	0.1	False	$\mathcal{N}(0, 0.1I_d)$
IFELSE+WHILE	32	0.25	0.1	True	$\mathcal{N}(0, 0.1I_d)$
2IF+IFELSE	16	0.25	0.2	True	$\mathcal{N}(0, 0.1I_d)$
WHILE+2IF+IFELSE	32	0.25	0.2	False	$\mathcal{N}(0, 0.1I_d)$

MDP Task Performance We include the following sets of hyperparameters when searching over the LEAPS program embedding space to maximize rewards in the MDP. We allow the search to run for 1000 CEM iterations, counting the search as a success when it achieves 10 consecutive CEM iterations of maximizing environment reward (solving the task) across 10 random environment start configurations.

CEM	S	σ	# Elites	Exp Decay	D_I
CLEANHOUSE	32	0.25	0.05	True	$\mathcal{N}(1, \mathbf{0})$
FOURCORNER	64	0.5	0.2	False	$\mathcal{N}(0, 0.1I_d)$
HARVESTER	32	0.5	0.1	True	$\mathcal{N}(0, I_d)$
MAZE	16	0.1	0.1	False	$\mathcal{N}(1, \mathbf{0})$
STAIRCLIMBER	32	0.25	0.05	True	$\mathcal{N}(0, 0.1I_d)$
TOPOFF	64	0.25	0.05	False	$\mathcal{N}(0, 0.1I_d)$

3.7.12.8 Random Search LEAPS Ablation

The random search LEAPS ablations (LEAPS-rand-8 and LEAPS-rand-64) replace the CEM search method for latent program synthesis with a simple random search method. Both use the full LEAPS model trained with all learning objectives. We sample an initial vector from an initial distribution D_I and add it to either 8 or 64 latent vector samples from a $\mathcal{N}(0, \sigma I_d)$ distribution. We then decode those vectors into programs and evaluate their rewards, and then report the rewards of the best-performing latent program from that population.

As such, the only parameters that we require are the initial sampling distribution and σ . We perform a grid search over the following for both LEAPS-rand-8 and LEAPS-rand-64.

- $\sigma: \{0.1, 0.25, 0.5\}$
- Initial distribution $D_I: \{\mathcal{N}(0, I_d), \mathcal{N}(0, 0.1I_d)\}$

Ground-Truth Program Reconstruction We report hyperparameters below for both random search methods on program reconstruction tasks.

LEAPS-rand-8	σ	D_I
WHILE	0.1	$\mathcal{N}(0, 0.1I_d)$
IFELSE+WHILE	0.5	$\mathcal{N}(0, 0.1I_d)$
2IF+IFELSE	0.5	$\mathcal{N}(0, 0.1I_d)$
WHILE+2IF+IFELSE	0.5	$\mathcal{N}(0, 0.1I_d)$

LEAPS-rand-64	σ	D_I
WHILE	0.5	$\mathcal{N}(0, 0.1I_d)$
IFELSE+WHILE	0.5	$\mathcal{N}(0, 0.1I_d)$
2IF+IFELSE	0.5	$\mathcal{N}(0, 0.1I_d)$
WHILE+2IF+IFELSE	0.5	$\mathcal{N}(0, 0.1I_d)$

MDP Task Performance We report hyperparameters below for both random search methods on Karel tasks.

LEAPS-rand-8	σ	D_I
CLEANHOUSE	0.5	$\mathcal{N}(0, 0.1I_d)$
FOURCORNER	0.5	$\mathcal{N}(0, 0.1I_d)$
HARVESTER	0.5	$\mathcal{N}(0, 0.1I_d)$
MAZE	0.25	$\mathcal{N}(0, 0.1I_d)$
STAIRCLIMBER	0.5	$\mathcal{N}(0, I_d)$
TOPOFF	0.25	$\mathcal{N}(0, 0.1I_d)$

	LEAPS-rand-64	σ	D_I
CLEANHOUSE	0.5	$\mathcal{N}(0, 0.1I_d)$	
FOURCORNER	0.25	$\mathcal{N}(0, 0.1I_d)$	
HARVESTER	0.5	$\mathcal{N}(0, 0.1I_d)$	
MAZE	0.1	$\mathcal{N}(0, 0.1I_d)$	
STAIRCLIMBER	0.25	$\mathcal{N}(0, 0.1I_d)$	
TOPOFF	0.5	$\mathcal{N}(0, 0.1I_d)$	

3.7.13 Computational Resources

For our experiments, we used both internal and cloud provider machines. Our internal machines are:

- M1: 40-vCPU Intel Xeon with 4 GTX Titan Xp GPUs
- M2: 72-vCPU Intel Xeon with 4 RTX 2080 Ti GPUs

The cloud instances that we used are either 128-thread AMD Epyc or 96-thread Intel Xeon based cloud instances with 4-8 NVIDIA Tesla T4 GPUs. Experiments were run in parallel across many CPUs whenever possible, thus requiring the high vCPU count machines.

The experiment costs (GPU memory/time) are as follows:

Learning Program Embedding Stage:

- LEAPS-P: 4.2GB/13hrs on either M1 or M2
- LEAPS-P+R: 4.2GB/44-54hrs on M2
- LEAPS-P+L: 8.7GB/26hrs on either M1 or M2
- LEAPS: 8.8GB/104hrs on M1, 8.8GB/58hrs on M2

Policy Learning Stage:

- CEM search: 0.8GB/4-10min (depends on the CEM population size and the number of iterations until convergence)
- DRL/DRL-abs/DRL-abs-t: 0.7-2GB/1hr per run with parallelization across 10 processes
- HRL/HRL-abs: 1-2GB/2.5hrs per run
- VIPER: 0.7GB/20-30 minutes (excluding the time for learning its teacher policy)

3.7.14 Toward Robotics Applications

One way to make the LEAPS framework applicable to robotics domains would be simultaneously learning perception modules and action controllers. Other possible solutions include incorporating program execution methods [11, 216, 287, 336, 160, 124] that are designed to allow program execution or designing DSLs that allow pre-training of perception modules and action controllers.

Also, the proposed framework share similarity with works in multi-task RL [216, 11, 294, 298, 281] and meta-learning [280, 202, 318, 314, 310, 244, 77, 139, 315, 45, 158, 212, 252, 193, 239, 49]. Specifically, the proposed framework learns a program embedding space from a distribution of tasks/programs. Once the program embedding space is learned, it can be reused to solve different tasks without retraining.

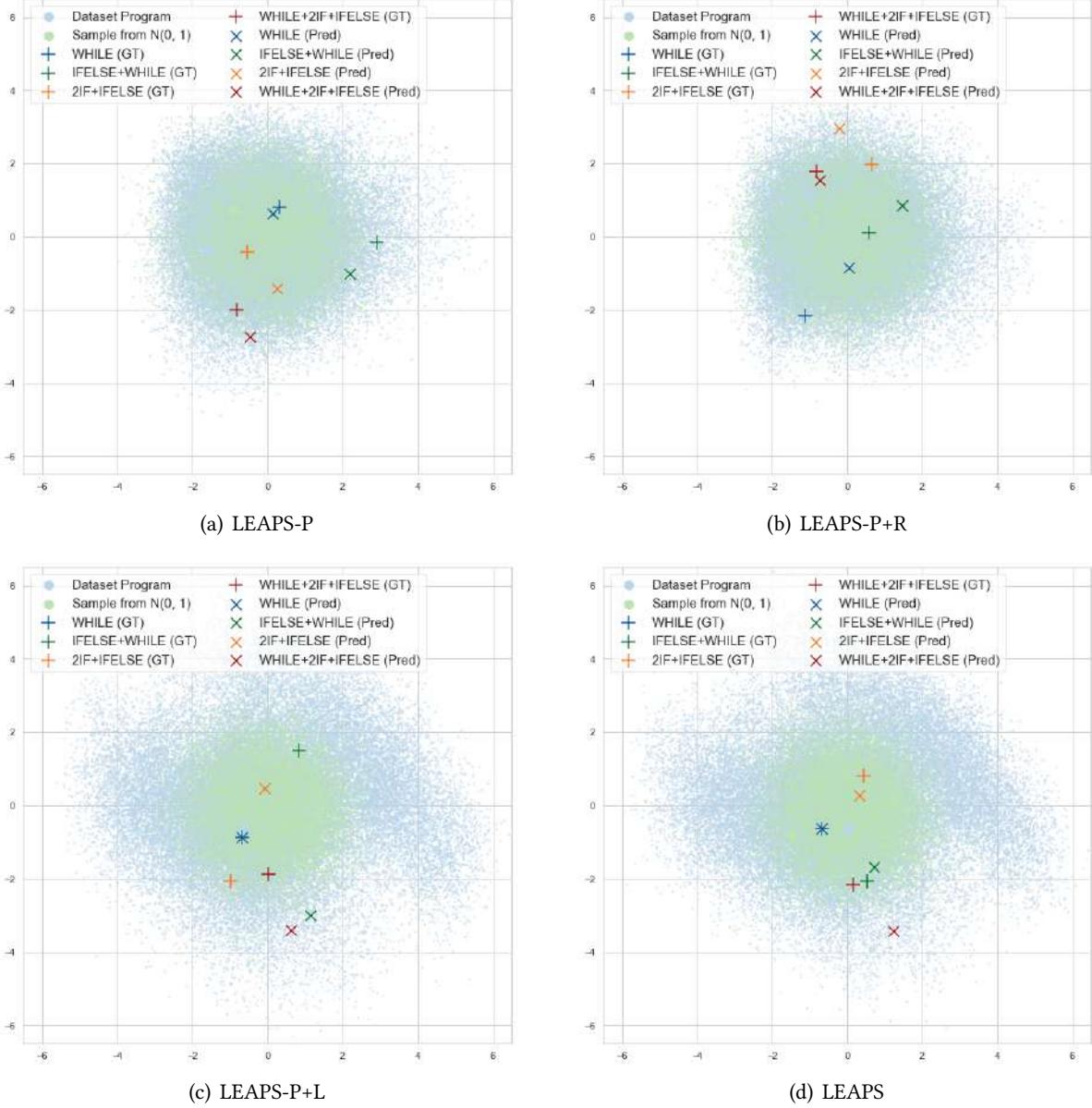


Figure 3.4: Visualizations of learned program embedding space. We perform dimensionality reduction with PCA to embed encoded programs from the training dataset, samples drawn from a normal distribution, programs from the testing dataset, and programs reconstructed by models to a 2D space. The shape of the latent training programs in the program embedding spaces learned by LEAPS-P and LEAPS-P+R are similar to a normal distribution, while in the program embedding spaces learned by LEAPS and LEAPS-P+L, the shape is more twisted, suggesting the effectiveness of the proposed latent behavior reconstruction objective. Moreover, the distances between pairs of ground-truth programs and their reconstructions are smaller in the program embedding space learned by LEAPS, highlighting the advantage of employing both of the two proposed behavior reconstruction objectives.

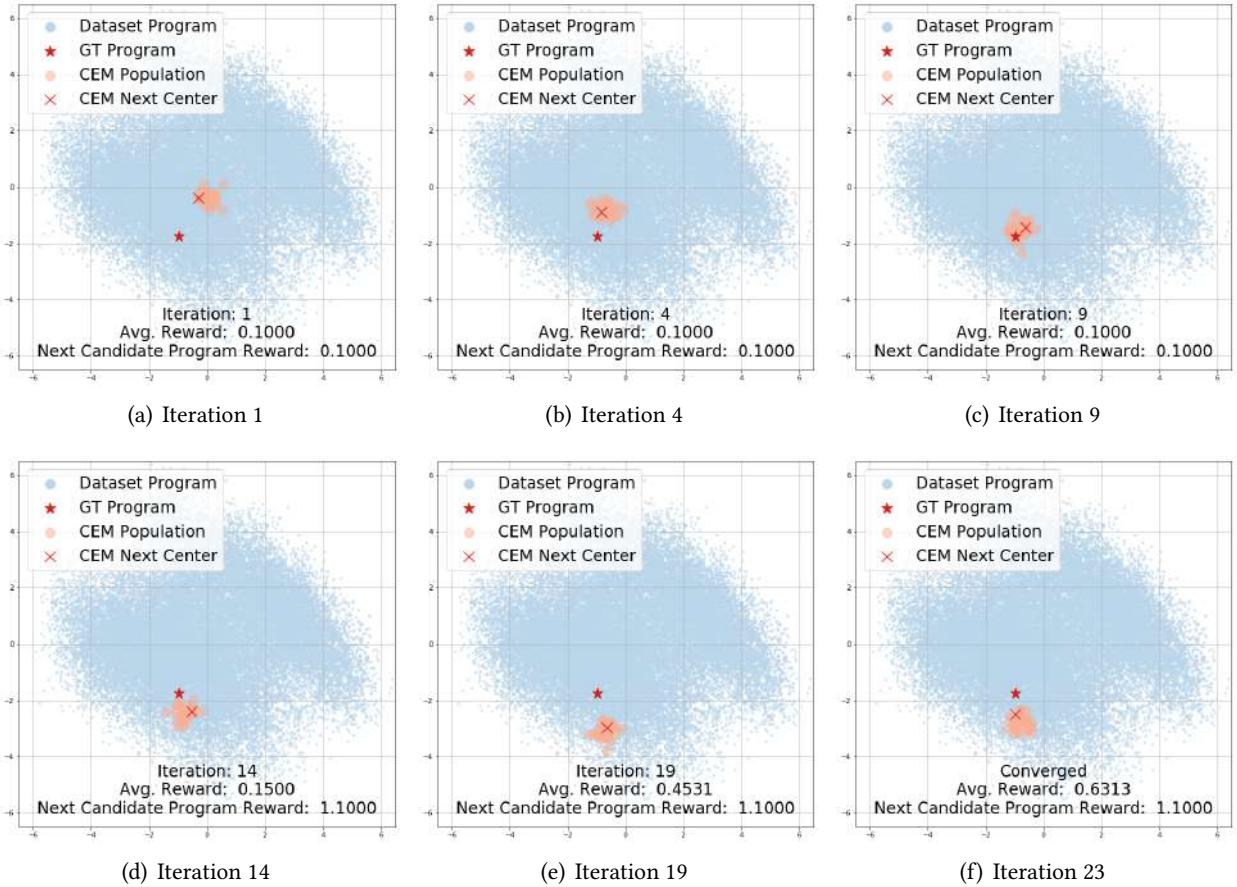


Figure 3.5: STAIRCLIMBER CEM Trajectory Visualization. Latent training programs from the training dataset, a ground-truth program for STAIRCLIMBER task, CEM populations, and CEM next candidate programs are embedded to a 2D space using PCA. Both the average reward of the entire population and the reward of the next candidate program (CEM Next Center) consistently increase as the number of iterations increase. Also, the CEM population gradually moves toward where the ground-truth program is located.

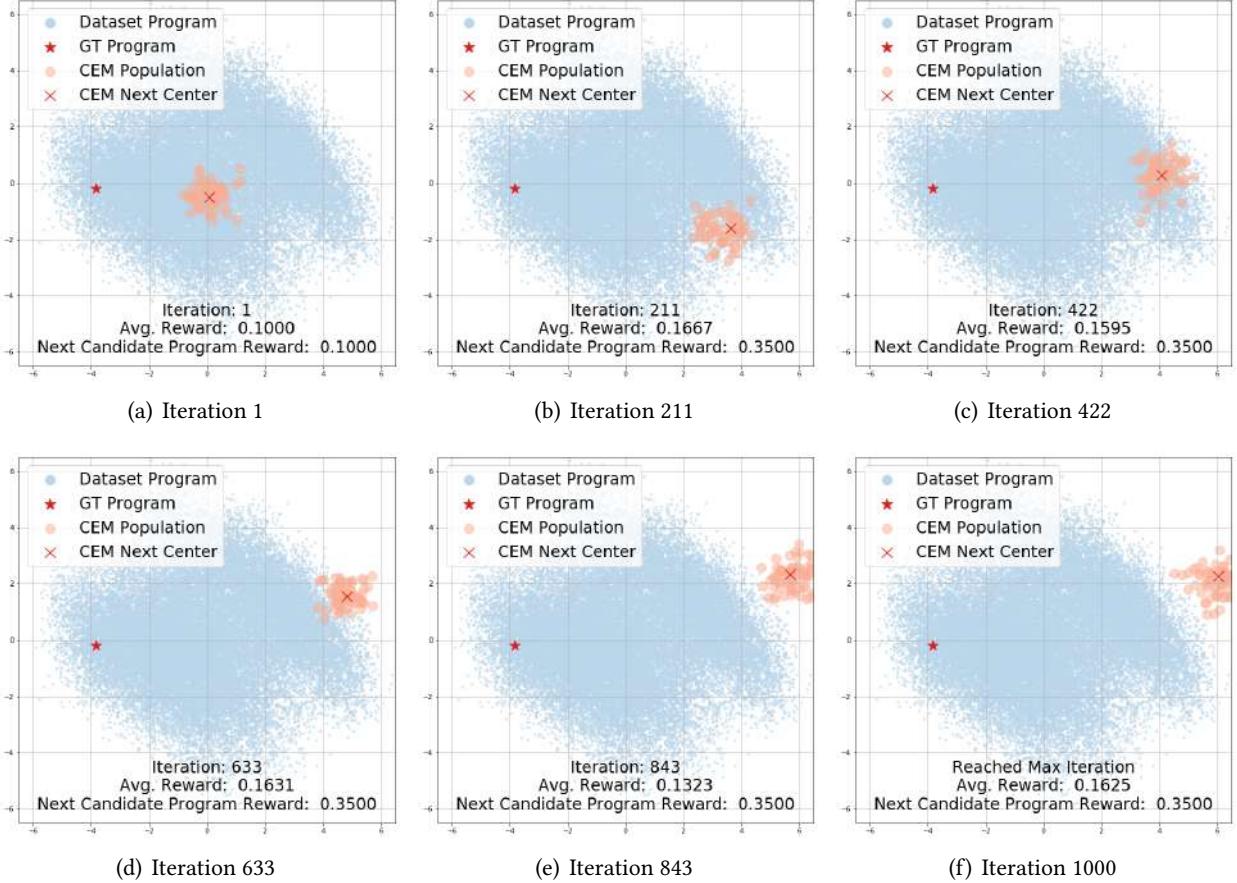


Figure 3.6: FOURCORNER CEM Trajectory Visualization. Latent training programs from the training dataset, a ground-truth program for the FOURCORNER task, CEM populations, and CEM next candidate programs are embedded to a 2D space using PCA. The CEM trajectory does not converge. The ground-truth program lies far away from the initial sampled distribution, which might contribute to the difficulty of converging.

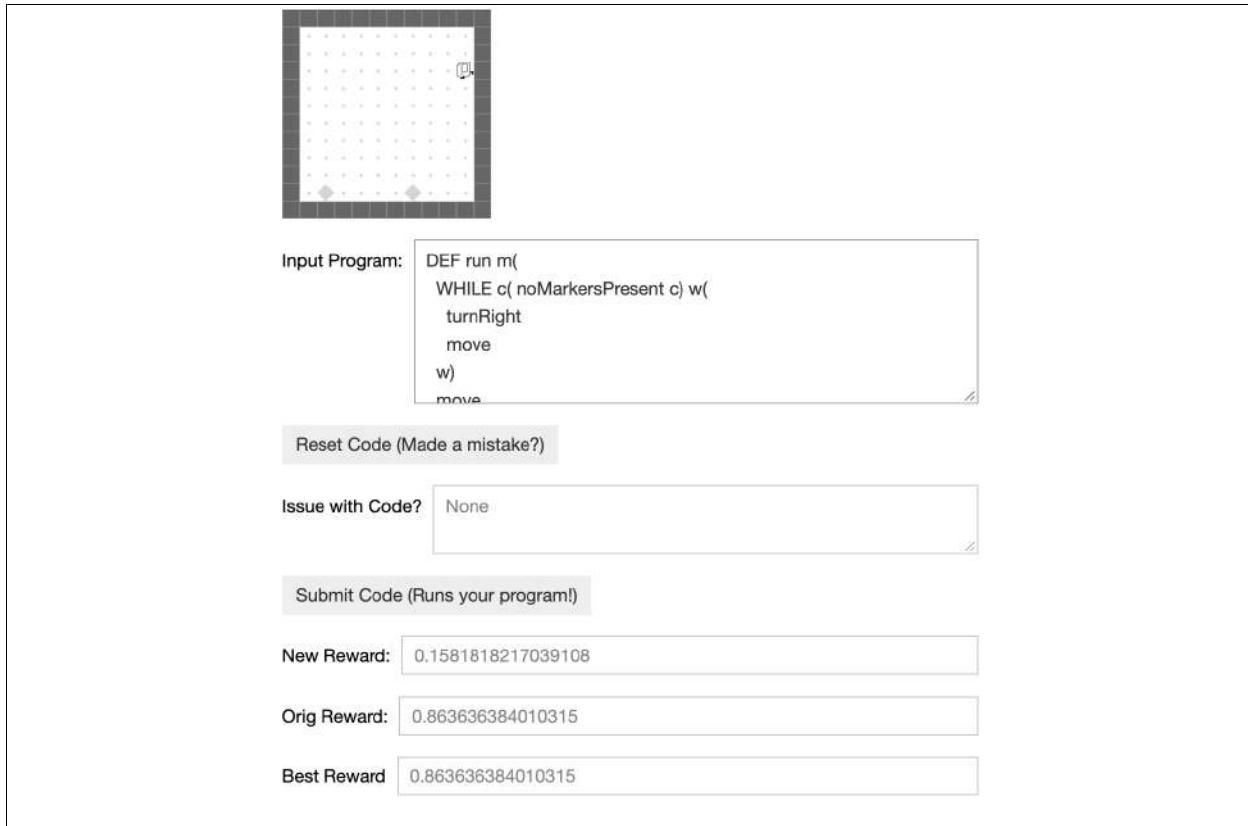


Figure 3.7: User Interface for the Human Debugging Interpretability Experiments. The top contains moving rollout visualizations of the current program in the “Input Program” box, which users are allowed to edit. “Input Program” will first contain the program synthesized by LEAPS. Syntax errors or other issues with code (such as the edit distance being too high) are displayed in the “Issue with Code?” box, the reward of the current inputted program is in the “New Reward” box, and the reward of the original program synthesized by LEAPS is in the “Orig Reward” box. The user’s best reward across all inputted programs is kept track of in the “Best Reward” box.

TOPOFF

LEAPS (REWARD=0.86)	3 EDITS (REWARD=1.0)	5 EDITS (REWARD=1.0)
<pre> DEF run m(WHILE c(noMarkersPresent c) w(turnRight move w) putMarker move WHILE c(noMarkersPresent c) w(turnRight move w) putMarker move WHILE c(noMarkersPresent c) w(turnRight move w) putMarker move WHILE c(noMarkersPresent c) w(turnRight move w) putMarker move WHILE c(noMarkersPresent c) w(turnRight move w) putMarker move WHILE c(noMarkersPresent c) w(turnRight move w) putMarker move WHILE c(noMarkersPresent c) w(turnRight move w) putMarker move m) </pre>	<pre> DEF run m(REPEAT R=9 r(WHILE c(noMarkersPresent c) w(IF c(frontIsClear c) i(move i) putMarker move r) WHILE c(noMarkersPresent c) w(turnRight move w) putMarker move WHILE c(noMarkersPresent c) w(turnRight move w) putMarker move WHILE c(noMarkersPresent c) w(turnRight move w) putMarker move WHILE c(noMarkersPresent c) w(turnRight move w) putMarker move m) </pre>	<pre> DEF run m(WHILE c(frontIsClear c) w(IF c(markersPresent c) i(putMarker i) move w) WHILE c(frontIsClear c) w(WHILE c(noMarkersPresent c) w(turnRight move w) putMarker move WHILE c(noMarkersPresent c) w(turnRight move w) putMarker move WHILE c(noMarkersPresent c) w(turnRight move w) putMarker move WHILE c(noMarkersPresent c) w(turnRight move w) putMarker move m) </pre>

Figure 3.8: **Human Debugging Experiment Example Programs (TopOFF)**. Example original and human-edited programs for each Karel task for edit distances 3 and 5.

FOURCORNER

LEAPS (REWARD=0.25)

3 EDITS (REWARD=1.0)

```
DEF run m{
    turnRight
    turnRight
    move
    turnRight
    REPEAT R=4 r(
    REPEAT R=9 r(
    move
    r)
    putMarker
    turnRight
    r)
    turnRight
m)
```

5 EDITS (REWARD=1.0)

```
DEF run m(
    turnRight
    turnRight
    move
    turnRight
    putMarker
REPEAT R=3 r(
REPEAT R=9 r(
move
r)
putMarker
turnRight
r)
m)
```

Figure 3.9: **Human Debugging Experiment Example Programs (FOURCORNER)**. Example original and human-edited programs for each Karel task for edit distances 3 and 5.

HARVESTER

LEAPS (REWARD=0.47)

```
DEF run m
  turnLeft
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  m)
```

3 EDITS (REWARD=0.77)

```
DEF run m{  
    turnLeft  
    turnLeft  
    REPEAT R=4 r(  
        pickMarker  
        move  
        pickMarker  
        move  
        pickMarker  
        move  
        pickMarker  
        move  
        pickMarker  
        move  
        pickMarker  
        move  
        turnLeft  
    r)  
    pickMarker  
    move  
    pickMarker  
    move  
    pickMarker  
    move  
    pickMarker  
    move  
    pickMarker  
    move  
    turnLeft  
    pickMarker  
    move  
    pickMarker  
    move  
    pickMarker  
    move  
    turnLeft  
    pickMarker  
    move  
    pickMarker  
    move  
    turnLeft  
    pickMarker  
    move  
    pickMarker  
    move  
    pickMarker  
    move  
    m)
```

5 EDITS (REWARD=0.89)

```
DEF run m()
  REPEAT R=3 r(
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    turnLeft
  r)
  REPEAT R=3 r(
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    turnLeft
  r)
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  turnLeft
  pickMarker
  move
  pickMarker
  move
  pickMarker
  move
  m)
```

Figure 3.10: **Human Debugging Experiment Example Programs (HARVESTER)**. Example original and human-edited programs for each Karel task for edit distances 3 and 5.

WHILE:

```
DEF run m(
    WHILE c( frontIsClear c) w(
        turnRight
        move
        pickMarker
        turnRight
        w)
    m)
```

IFELSE+WHILE:

```
DEF run m(
    IFELSE c( markersPresent c) i(
        move turnRight
        i) ELSE e(
            move
            e)
        move
        move
    WHILE c( leftIsClear c) w(
        turnLeft
        w)
    m)
```

WHILE+2IF+IFELSE:

```
DEF run m(
    WHILE c( leftIsClear c) w(
        turnLeft
        w)
    IF c( frontIsClear c) i(
        putMarker move
        i)
    move
    IF c( rightIsClear c) i(
        turnRight
        move
        i)
    IFELSE c( frontIsClear c) i(
        move
        i) ELSE e(
            turnLeft move
            e)
    m)
```

CLEANHOUSE:

```
DEF run m(
    WHILE c( noMarkersPresent c) w(
        IF c( leftIsClear c) i(
            turnLeft
            i)
        move
        IF c( markersPresent c) i(
            pickMarker
            i)
        w)
    m)
```

STAIRCLIMBER:

```
DEF run m(
    WHILE c( noMarkersPresent c) w(
        turnLeft
        move
        turnRight
        move
        w)
    m)
```

2IF+IFELSE:

```
DEF run m(
    IF c( frontIsClear c) i(
        putMarker
        i)
    move
    IF c( rightIsClear c) i(
        move
        i)
    IFELSE c( frontIsClear c) i(
        move
        i) ELSE e(
            move
            e)
    m)
```

TOPOFF:

```
DEF run m(
    WHILE c( frontIsClear c) w(
        IF c( markersPresent c) i(
            putMarker
            i)
        move
        w)
    m)
```

FOURCORNER:

```
DEF run m(
    WHILE c( noMarkersPresent c) w(
        WHILE c( frontIsClear c) w(
            move
            w)
        IF c( noMarkersPresent c) i(
            (
            putMarker
            turnLeft
            move
            i)
        w)
    m)
```

MAZE:

```
DEF run m(
    WHILE c( noMarkersPresent c) w(
        IFELSE c( rightIsClear c) i(
            (
            turnRight
            i) ELSE e(
                WHILE c( not c(
                    frontIsClear
                    c) c) w(
                    turnLeft
                    w)
            e)
        move
        w)
    m)
```

HARVESTER:

```
DEF run m(
    WHILE c( markersPresent c) w(
        WHILE c( markersPresent c) w(
            pickMarker
            move
            w)
        turnRight
        move
        turnLeft
    WHILE c( markersPresent c) w(
        pickMarker
        move
        w)
        turnLeft
        move
        turnRight
        w)
    m)
```

Figure 3.11: **Ground-Truth Test and Karel Programs.** Here we display ground-truth test set programs used for reconstruction experiments and example ground-truth programs that we write which can solve the Karel tasks (there are an infinite number of programs that can solve each task). Conditionals are enclosed in `c(c)`, while loops are enclosed in `w(w)`, if statements are enclosed in `i(i)`, and the main program is enclosed in `DEF run m(m)`.

Naïve	
WHILE	IFELSE+WHILE
<pre>DEF run m{ WHILE c(frontIsClear c) w(turnRight move pickMarker turnRight w) m)</pre>	<pre>DEF run m{ move move move turnLeft turnLeft m)</pre>
2IF+IFELSE	
<pre>DEF run m{ putMarker move move move m)</pre>	<pre>DEF run m(turnLeft putMarker move move WHILE c(markersPresent c) w(pickMarker pickMarker pickMarker w) m)</pre>
LEAPS-P	
WHILE	IFELSE+WHILE
<pre>DEF run m(IF c(frontIsClear c) i(turnRight move pickMarker turnRight i) m)</pre>	<pre>DEF run m(IFELSE c(rightIsClear c) i(move i) ELSE e(move e) move move IF c(leftIsClear c) i(turnLeft i) m)</pre>
2IF+IFELSE	
<pre>DEF run m(IFELSE c(not c(frontIsClear c) c) i(move i) ELSE e(putMarker move e) move move m)</pre>	<pre>DEF run m(WHILE c(leftIsClear c) w(turnLeft w) putMarker move move turnRight move move m)</pre>
LEAPS-P+R	
WHILE	IFELSE+WHILE
<pre>DEF run m(WHILE c(rightIsClear c) w(WHILE c(frontIsClear c) w(turnRight move pickMarker turnRight w) w) m)</pre>	<pre>DEF run m(REPEAT R=1 r(move r) REPEAT R=2 r(move r) m)</pre>
2IF+IFELSE	
<pre>DEF run m(IFELSE c(not c(frontIsClear c) c) i(move i) ELSE e(putMarker e) IFELSE c(rightIsClear c) i(move i) ELSE e(move e) IF c(rightIsClear c) i(move i) move m)</pre>	<pre>DEF run m(WHILE c(leftIsClear c) w(turnLeft w) putMarker move move turnRight move move m)</pre>

Figure 3.12: **Example program reconstruction task programs generated by naïve, LEAPS-P, and LEAPS-P+R.** The programs that achieve the highest reward while being representative of programs generated by most seeds are shown. The naïve program synthesis baseline usually generates the simplest programs, with fewer conditional statements and loops than the LEAPS ablations. Notably, it fails to generate IFELSE statements on these examples, while LEAPS has no problem doing so.

LEAPS-P+L	
WHILE	IFELSE+WHILE
<pre>DEF run m(WHILE c(frontIsClear c) w(turnRight move pickMarker turnRight w) m)</pre>	<pre>DEF run m(move move move WHILE c(leftIsClear c) w(turnLeft w) m)</pre>
2IF+IFELSE	
<pre>DEF run m(IFELSE c(frontIsClear c) i(REPEAT R=0 r{ turnRight r) putMarker move i) ELSE e(move e) move move m)</pre>	<pre>DEF run m(WHILE c(leftIsClear c) w(turnLeft w) WHILE c(leftIsClear c) w(turnLeft w) WHILE c(leftIsClear c) w(turnLeft w) IF c(frontIsClear c) i(putMarker move i) move move m)</pre>
LEAPS	
WHILE	IFELSE+WHILE
<pre>DEF run m(WHILE c(frontIsClear c) w(turnRight move pickMarker turnRight w) m)</pre>	<pre>DEF run m(IFELSE c(not c(noMarkersPresent c) c) i(move turnRight i) ELSE e(move e) REPEAT R=2 r{ move r) WHILE c(leftIsClear c) w(turnLeft w) m)</pre>
2IF+IFELSE	
<pre>DEF run m(IFELSE c(frontIsClear c) i(putMarker move i) ELSE e(move e) IF c(rightIsClear c) i(move i) move m)</pre>	<pre>DEF run m(WHILE c(leftIsClear c) w(turnLeft w) IF c(frontIsClear c) i(putMarker move i) move move m)</pre>
WHILE+2IF+IFELSE	

Figure 3.13: **Example program reconstruction task programs generated by LEAPS-P+L and LEAPS.** The programs that achieve the highest reward while being representative of programs generated by most seeds are shown. The naïve program synthesis baseline usually generates the simplest programs, with fewer conditional statements and loops than the LEAPS ablations. Notably, it fails to generate IFELSE statements on these examples, while LEAPS has no problem doing so.

LEAPS Karel Programs

STAIRCLIMBER

```

DEF run m(
  WHILE c( noMarkersPresent c)
    w(
      turnRight
      move
      w)
  WHILE c( rightIsClear c) w(
    turnLeft
    w)
  m)

```

TOPOFF

CLEANHOUSE

```

DEF run m(
  WHILE c( noMarkersPresent c)
    w(
      turnRight
      move
      move
      turnLeft
      turnRight
      pickMarker
      w)
    turnLeft
    turnRight
  m)

```

FOURCORNER

```

DEF run m(
    turnRight
    move
    turnRight
    turnRight
    turnRight
    WHILE c( frontIsClear c) w(
        move
        w)
    turnRight
    putMarker
    WHILE c( frontIsClear c) w(
        move
        w)
    turnRight
    putMarker
    WHILE c( frontIsClear c) w(
        move
        w)
    turnRight
    putMarker
    WHILE c( frontIsClear c) w(
        move
        w)
    turnRight
    putMarker
    m)

```

MAZE

```

DEF run m(
  IF c( frontIsClear c) i(
    turnLeft
    i)
  WHILE c( noMarkersPresent c)
    w(
      turnRight
      move
      w)
    m)

```

HARVESTER

```
DEF run m(
    turnLeft
    turnLeft
    pickMarker
    move
    pickMarker
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    turnLeft
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    turnLeft
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    turnLeft
    pickMarker
    move
    pickMarker
    move
    pickMarker
    move
    m)
```

Figure 3.14: Example Karel programs generated by LEAPS. The programs that achieved the best reward out of all seeds are shown.

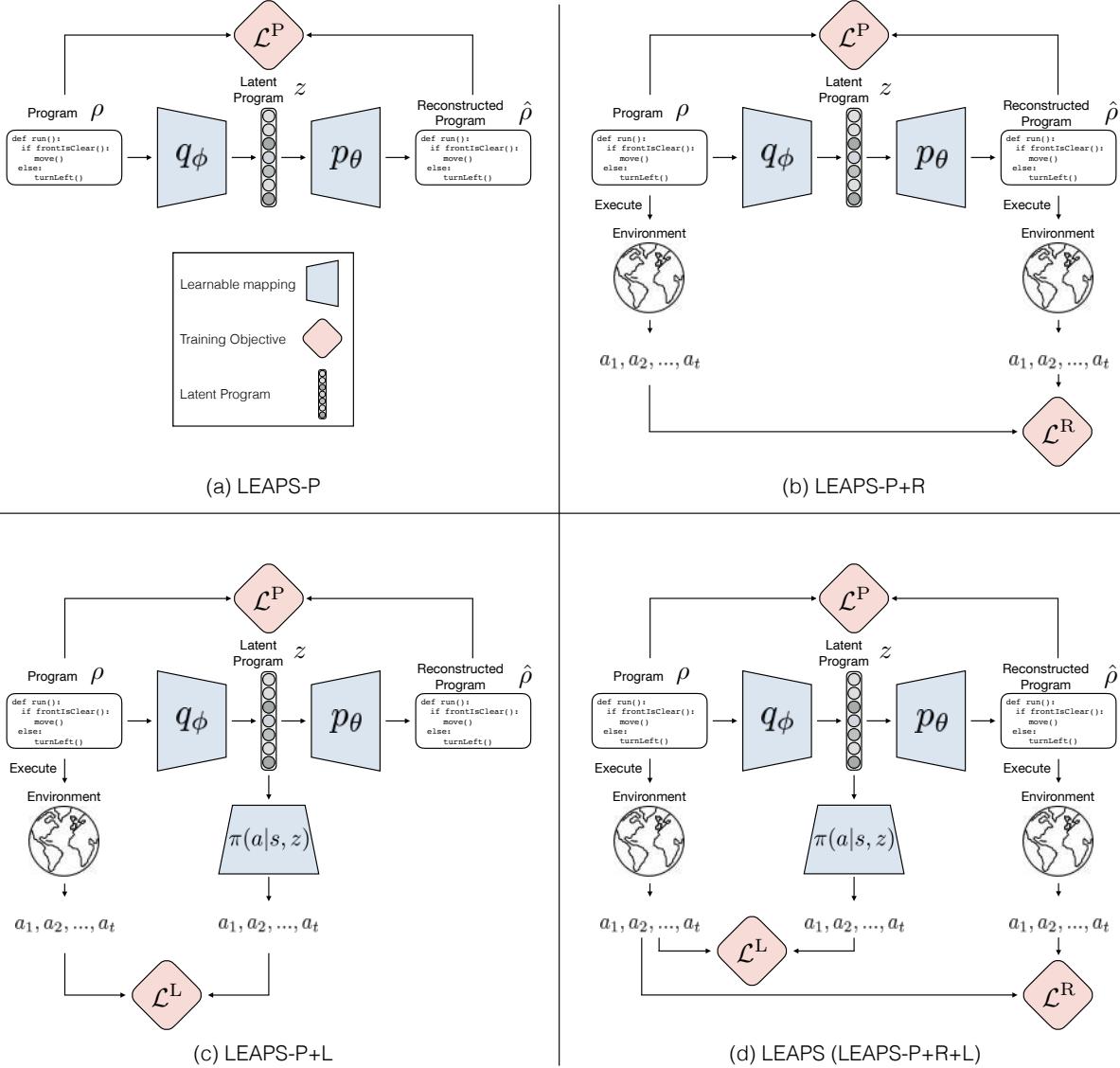


Figure 3.15: LEAPS Variations Illustrations. Blue trapezoids represent the modules whose parameters are being learned in the learning program embedding stage. Red diamonds represent the learning objectives. Gray rounded rectangle represent latent programs (*i.e.* program embeddings), which are vectors. (a) LEAPS-P: the simplest ablation of LEAPS, in which the program embedding space is learned by only optimizing the program reconstruction loss \mathcal{L}^P . (b) LEAPS-P+R: an ablation of LEAPS which optimizes both the program reconstruction loss \mathcal{L}^P and the program behavior reconstruction loss \mathcal{L}^R . (c) LEAPS-P+L: an ablation of LEAPS which optimizes both the program reconstruction loss \mathcal{L}^P and the latent behavior reconstruction loss \mathcal{L}^L . (d) LEAPS (LEAPS-P+R+L): our proposed framework that optimizes all the proposed objectives.

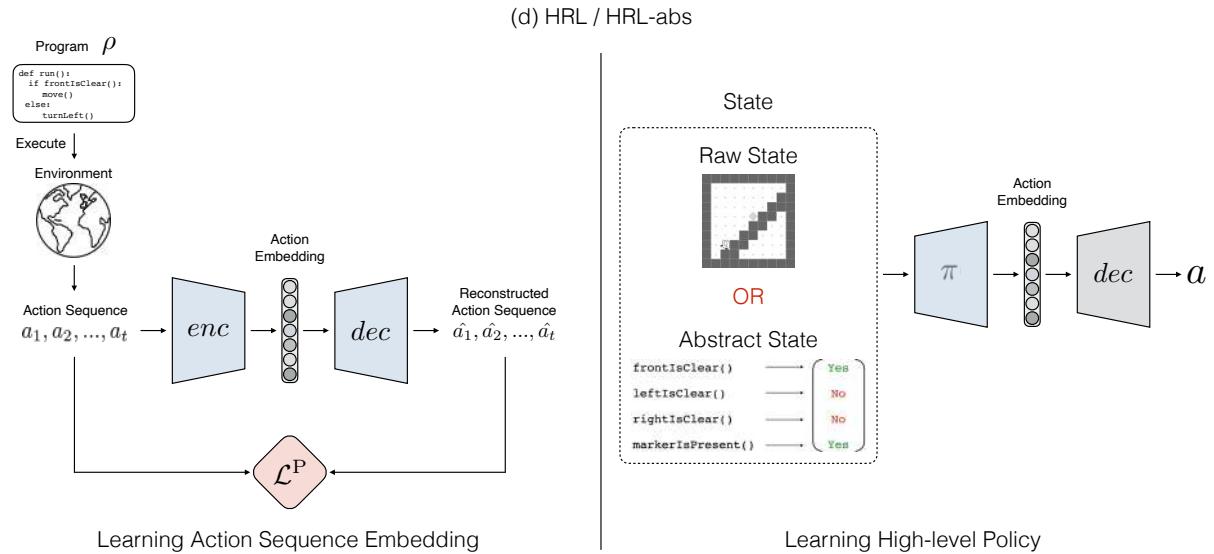
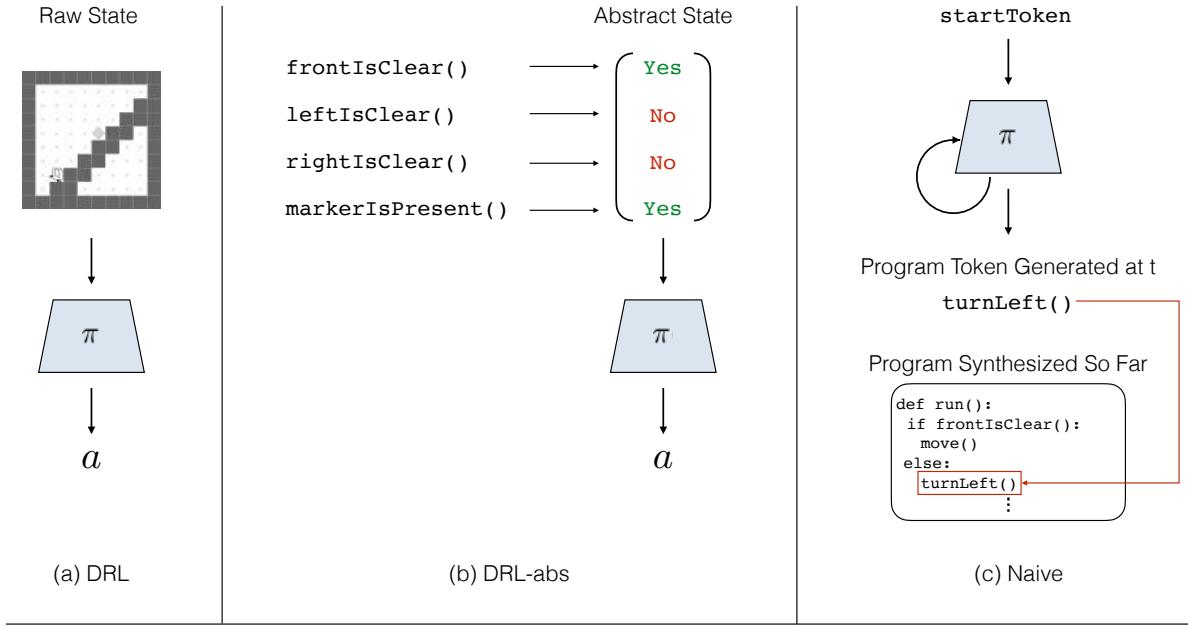


Figure 3.16: **Baseline Methods Illustrations.** (a) DRL: a DRL policy that takes raw state input (*i.e.* a Karel grid represented as a $W \times H \times 12$ binary tensor as there are 12 possible states for each grid square). (b) DRL-abs: a DRL policy that takes abstract state input, containing a vector of returned values of perceptions, *e.g.* `frontIsClear() == true` and `markerIsPresent() == false`. (c) Naive: a naïve program synthesis baseline that learns to directly synthesize a program from scratch by recurrently predicting a sequence of program tokens. (d) HRL/HRL-abs: a hierarchical RL baseline in which a VAE, consisting of a encoder enc and a decoder dec , is first trained to reconstruct action sequences from program execution traces used by LEAPS. Once the action embedding space is learned, it employs a high-level policy π that learns from scratch to solve task by predicting a distribution in the learned action embedding space. Note that the parameters of the decoder dec are frozen (represented in gray) when the high-level policy is learning. The HRL policy takes raw state input (same as the DRL baseline) and the HRL-abs policy takes abstract state input (same as the DRL-abs baseline).

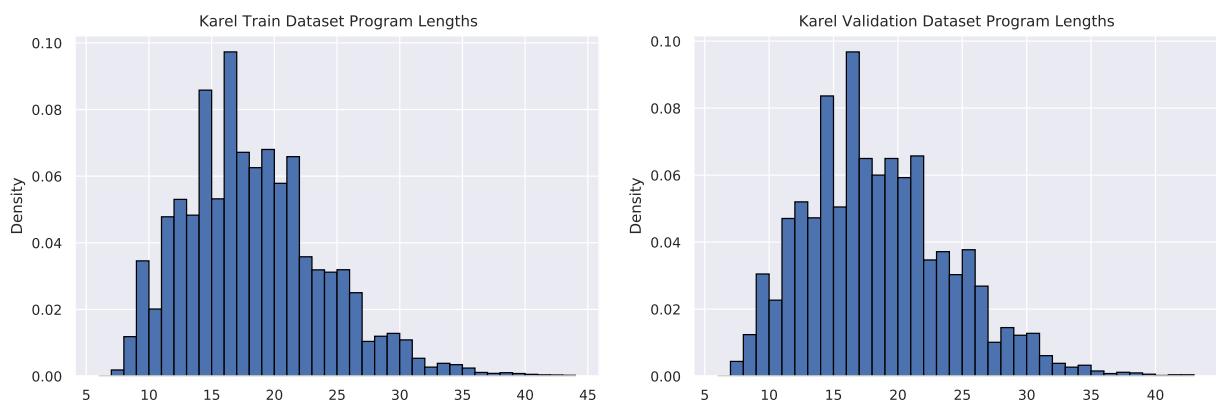


Figure 3.17: Histograms of the program length (*i.e.* number of program tokens) in the training and validation datasets.

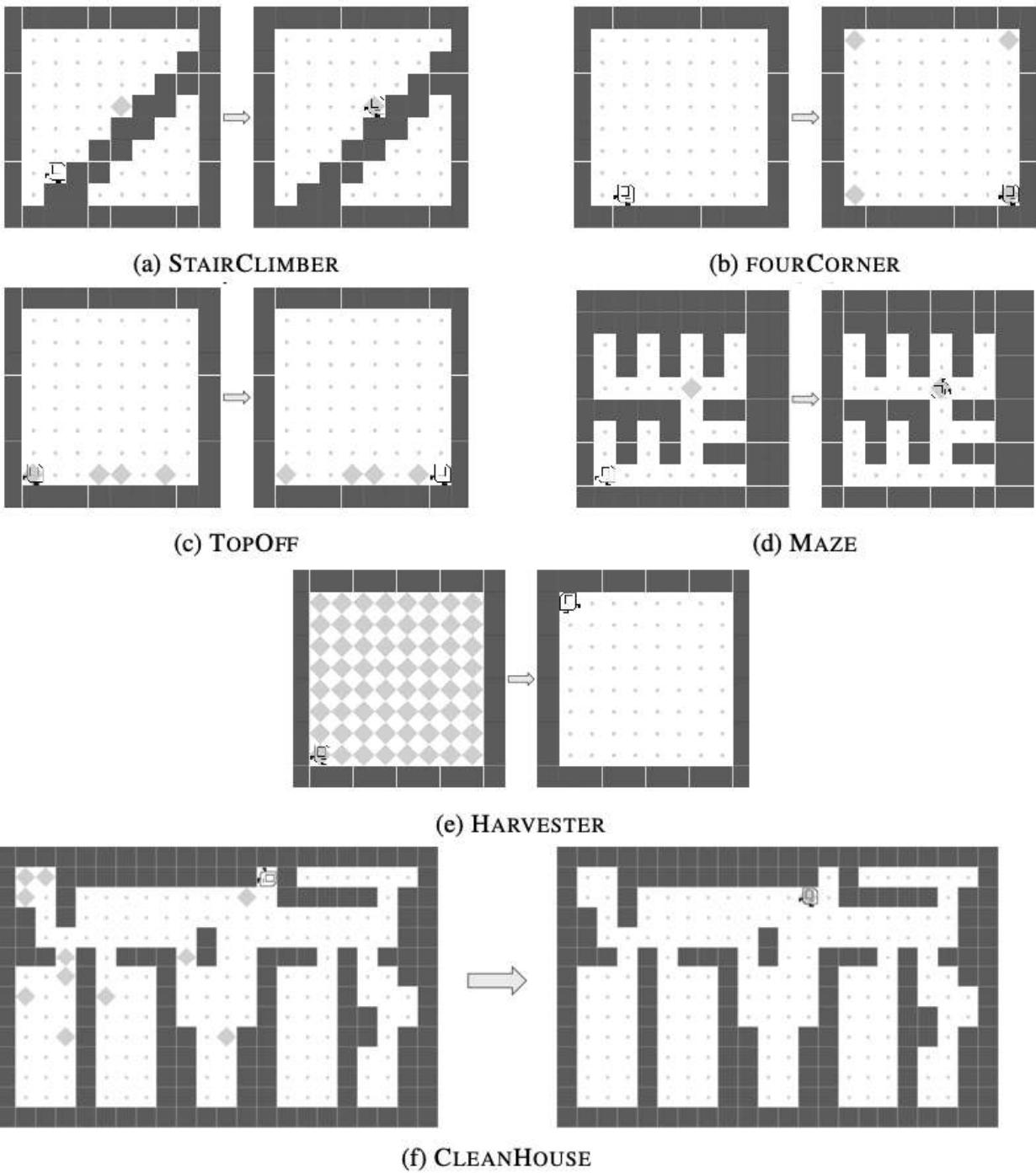
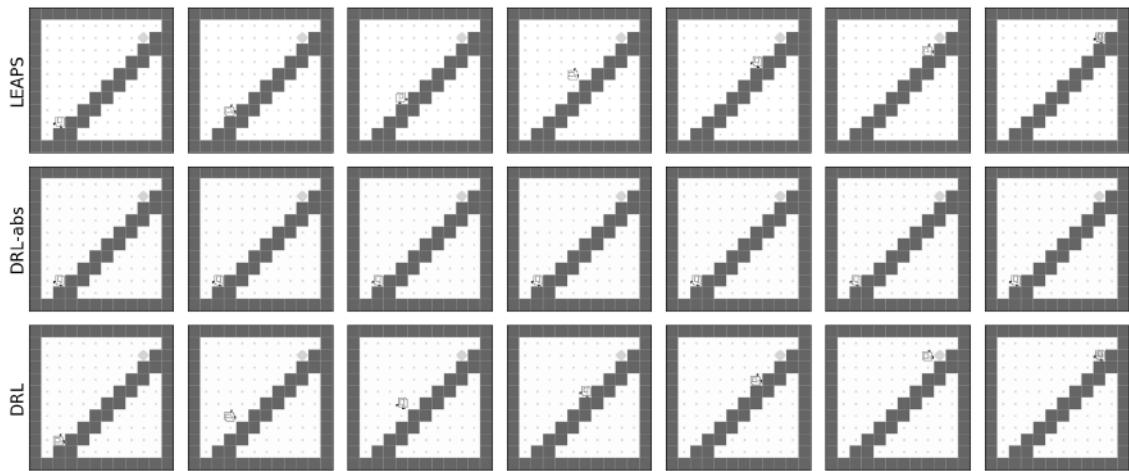
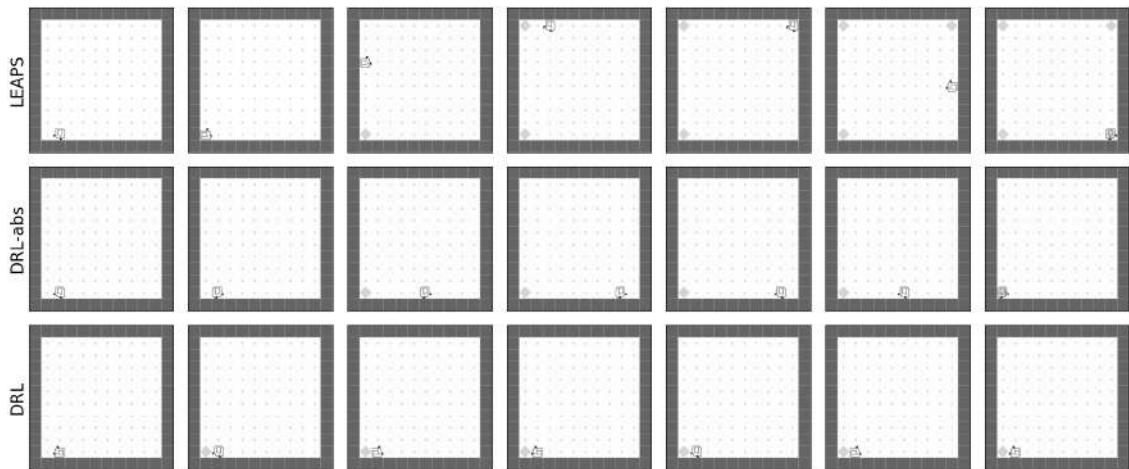


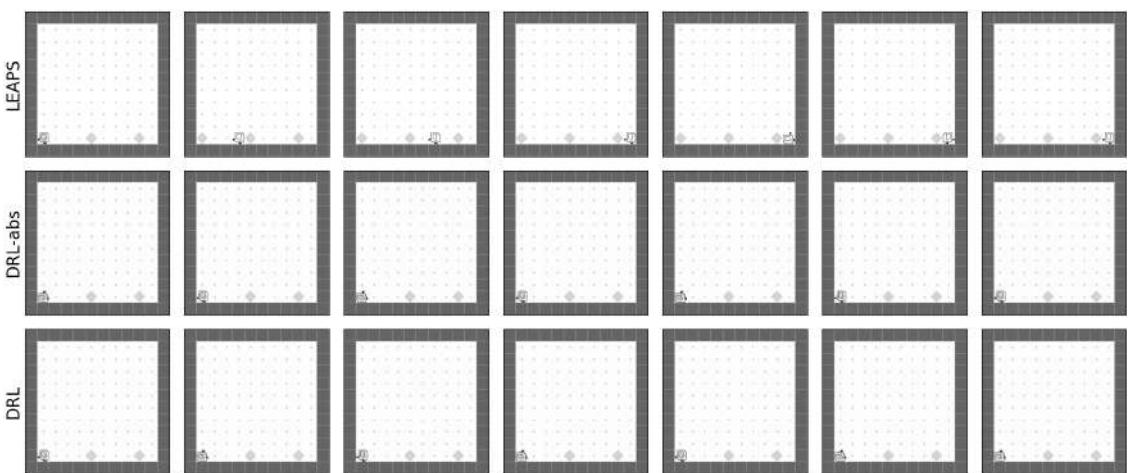
Figure 3.18: Example of initial configurations and their ideal end states of the Karel tasks. Note that we show only one example of initial configuration and its ideal end state pair for each task. However, markers, walls and agent's position are randomized in initial configurations depending upon task. Please see section 3.7.11 for more details.



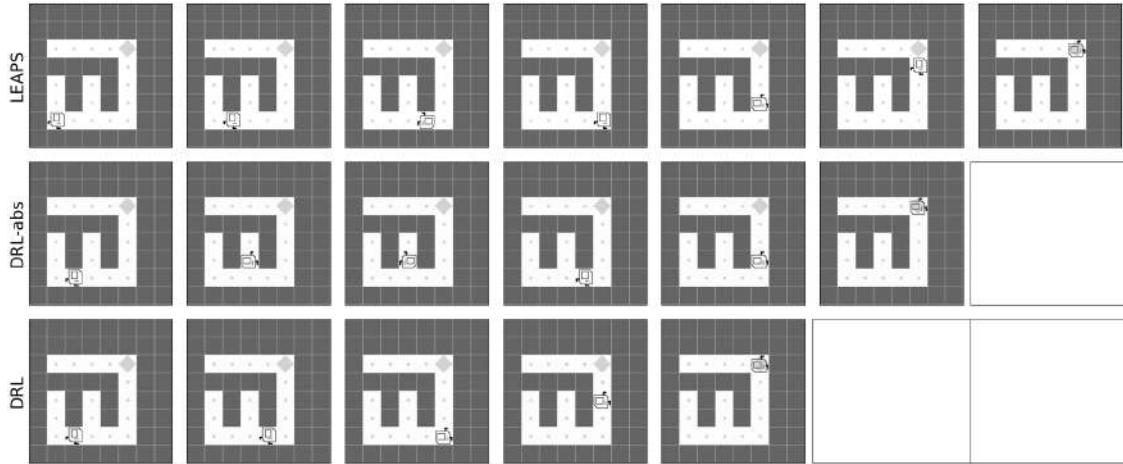
(a) STAIRCLIMBER: LEAPS and DRL are able to climb the stairs, DRL-abs is unable to do so.



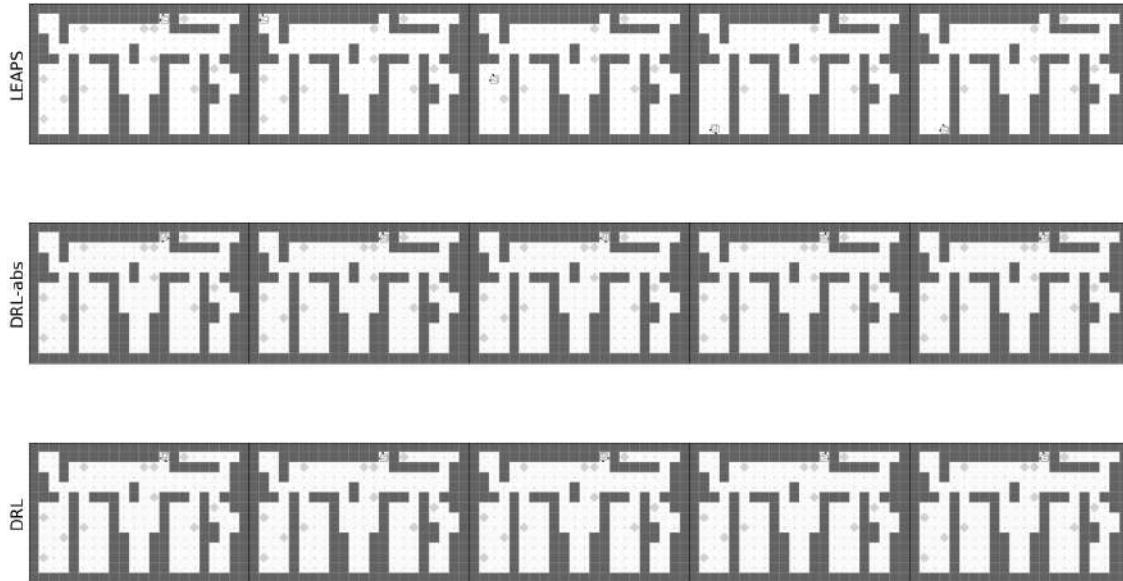
(b) FOURCORNER: In this example, LEAPS generates a program which is able to completely solve the task. Both DRL methods learn to only place one single marker in the left bottom corner.



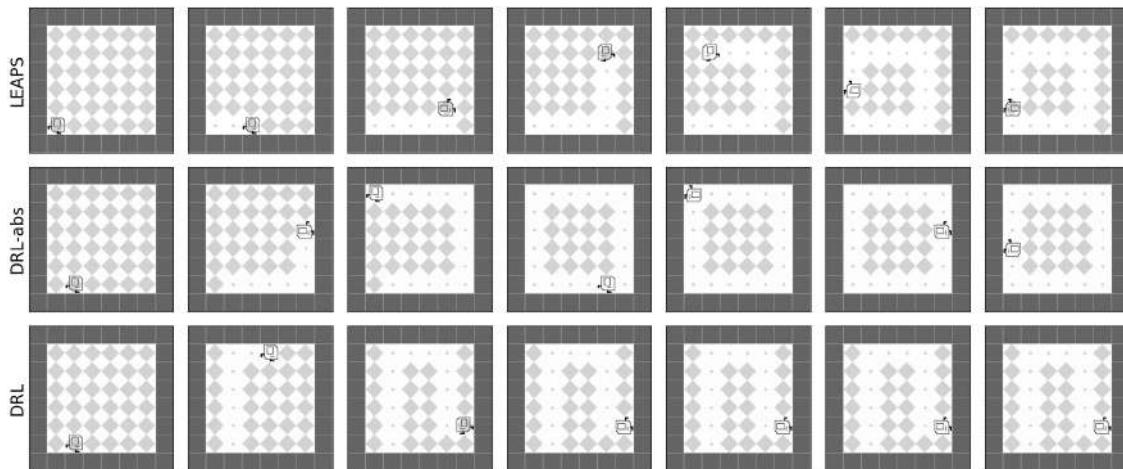
(c) TopOff: Here, LEAPS generates a program that solves the task by “topping off” each marker. Both DRL methods only learn to top off the initial marker.



(d) MAZE: All three methods are able to solve the task.



(e) CLEANHOUSE: While both DRL methods learn no meaningful behaviors (generally just spinning around in place), LEAPS generates a program that is able to navigate to and clean the leftmost room.



(f) HARVESTER: All three methods make partial progress on HARVESTER.

Figure 3.19: **Karel Rollout Visualizations.** Example rollouts for LEAPS, DRL-abs, and DRL for each task.

Part III

Primitive Skill Acquisition

Chapter 4

Meta-Learning on Multimodal Task Distributions

4.1 Introduction

Humans make effective use of prior knowledge to acquire new skills rapidly. When the skill of interest is related to a wide range of skills that one have mastered before, we can recall relevant knowledge of prior skills and exploit them to accelerate the new skill acquisition procedure. For example, imagine that we are learning a novel snowboarding trick with knowledge of basic skills about snowboarding, skiing, and skateboarding. We accomplish this feat quickly by exploiting our basic snowboarding knowledge together with inspiration from our skiing and skateboarding experience.

Can machines likewise quickly master a novel skill based on a variety of related skills they have already acquired? Recent advances in meta-learning [310, 79, 70] have attempted to tackle this problem. They offer machines a way to rapidly adapt to a new task using few samples by first learning an internal representation that matches similar tasks. Such representations can be learned by considering a distribution over similar tasks as the training data distribution. Model-based (*i.e.* RNN-based) meta-learning approaches [70, 318, 202, 193] propose to recognize the task identity from a few sample data, use the task identity to adjust a model’s state (*e.g.* RNN’s internal state or an external memory) and make the appropriate predictions with the adjusted model. Those methods demonstrate good performance at the expense of having to hand-design architectures, yet the optimal strategy of designing a meta-learner for arbitrary tasks may not always

be obvious to humans. On the other hand, model-agnostic meta-learning frameworks [77, 81, 139, 158, 94, 212, 254, 252] seek an initialization of model parameters that a small number of gradient updates will lead to superior performance on a new task. With the flexibility in the model choices, these frameworks demonstrate appealing performance on a variety of domains, including regression, image classification, and reinforcement learning.

While most of the existing model-agnostic meta-learners rely on a single initialization, different tasks sampled from a complex task distributions can require substantially different parameters, making it difficult to find a single initialization that is close to all target parameters. If the task distribution is multimodal with disjoint and far apart modes (e.g. snowboarding, skiing), one can imagine that a set of separate meta-learners with each covering one mode could better master the full distribution. However, associating each task with one of the meta-learners not only requires additional task identity information, which is often not available or could be ambiguous when the modes are not clearly disjoint, but also disables transferring knowledge across different modes of the task distribution. To overcome this issue, we aim to develop a meta-learner that is able to acquire mode-specific prior parameters and adapt quickly given tasks sampled from a multimodal task distribution.

To this end, we leverage the strengths of the two main lines of existing meta-learning techniques: model-based and model-agnostic meta-learning. Specifically, we propose to augment MAML [77] with the capability of generalizing across a multimodal task distribution. Instead of learning a single initialization point in the parameter space, we propose to first compute the task identity of a sampled task by examining task related data samples. Given the estimated task identity, our model then performs modulation to condition the meta-learned initialization on the inferred task mode. Then, with these modulated parameters as the initialization, a few steps of gradient-based adaptation are performed towards the target task to progressively improve its performance. An illustration of our proposed framework is shown in Figure 4.1.

To investigate whether our method can acquire meta-learned prior parameters by learning tasks sampled from multimodal task distributions, we design and conduct experiments on a variety of domains, including regression, image classification, and reinforcement learning. The results demonstrate the effectiveness of our approach against other systems. A further analysis has also shown that our method learns to identify task modes without extra supervision.

The main contributions of this paper are three-fold as follows:

- We identify and empirically demonstrate the limitation of having to rely on a single initialization in a family of widely used model-agnostic meta-learners.
- We propose a framework together with an algorithm to address this limitation. Specifically, it generates a set of meta-learned prior parameters and adapts quickly given tasks from a multimodal task distribution leveraging both model-based and model-agnostic meta-learning.
- We design a set of multimodal meta-learning problems and demonstrate that our model offers a better generalization ability in a variety of domains, including regression, image classification, and reinforcement learning.

4.2 Related Work

The idea of empowering the machines with the capability of *learning to learn* [297] has been widely explored by the machine learning community. To improve the efficiency of handcrafted optimizers, a flurry of recent works has focused on learning to optimize a learner model. Pioneered by [261, 28], optimization algorithms with learned parameters have been proposed, enabling the automatic exploitation of the structure of learning problems. From a reinforcement learning perspective, [165] represents an optimization algorithm as a learning policy. [13] trains LSTM optimizers to learn update rules from the gradient history, and [244]

trains a meta-learner LSTM to update a learner’s parameters. Similar approach for continual learning is explored in [313].

Recently, investigating how we can replicate the ability of humans to learn new concepts from one or a few instances, known as *few-shot learning*, has drawn people’s attention due to its broad applicability to different fields. To classify images with few examples, metric-based meta-learning frameworks have been proposed [143, 310, 280, 274, 288, 217, 45], which strive to learn a metric or distance function that can be used to compare two different samples effectively. Recent works along this line [217, 341, 158] share a conceptually similar idea with us and seek to perform task-specific adaptation with different type transformations. Due to the limited space, we defer the detailed discussion to Appendix (Section 4.7). While impressive results have been shown, it is nontrivial to adopt them for complex tasks such as acquiring robotic skills using reinforcement learning [118, 170, 133, 242, 98, 104, 160].

On the other hand, instead of learning a metric, model-based (*i.e.* RNN-based) meta-learning models learn to adjust model states (*e.g.* a state of an RNN [193, 70, 317] or external memory [256, 202]) using a training dataset and output the parameters of a learned model or the predictions given test inputs. While these methods have the capacity to learn any mapping from datasets and test samples to their labels, they could suffer from overfitting and show limited generalization ability [79].

Model-agnostic meta-learners [77, 81, 139, 158, 94, 212, 254, 252] are agnostic to concrete model configurations. Specifically, they aim to learn a parameter initialization under a certain task distribution, that aims to provide a favorable inductive bias for fast gradient-based adaptation. With its model agnostic nature, appealing results have been shown on a variety of learning problems. However, assuming tasks are sampled from a concentrated distribution and pursuing a common initialization to all tasks can substantially limit the performance of such methods on multimodal task distributions where the center in the task space becomes ambiguous.

In this paper, we aim to develop a more powerful model-agnostic meta-learning framework which is able to deal with complex multimodal task distributions. To this end, we propose a framework, which first identifies the mode of sampled tasks, similar to model-based meta-learning approaches, and then it modulates the meta-learned prior parameters to make the model better fit to the identified mode. Finally, the model is fine-tuned on the target task rapidly through gradient steps.

4.3 Preliminaries

The goal of meta-learning is to quickly learn task-specific functions that map between input data and the desired output $(x_k, y_k)_{k=1}^{K_t}$ for different tasks t , where the number of data K_t is small. A task is defined by the underlying data generating distribution $\mathcal{P}(X)$ and a conditional probability $\mathcal{P}_t(Y | X)$. For instance, we consider five-way image classification tasks with x_k to be images and y_k to be the corresponding labels, sampled from a task distribution. The data generating distribution is unimodal if it contains classification tasks that belong to a single input and label domain (e.g. classifying different combination of digits). A multimodal counterpart therefore contains classification tasks from multiple different input and label domains (e.g. classifying digits vs. classifying birds). We denote the later distribution of tasks to be the *multimodal task distribution*.

In this paper, we aim to rapidly adapt to a novel task sampled from a multimodal task distribution. We consider a target dataset \mathcal{D} consisting of tasks sampled from a multimodal distribution. The dataset is split into meta-training and meta-testing sets, which are further divided into task-specific training $\mathcal{D}_{\mathcal{T}}^{\text{train}}$ and validation $\mathcal{D}_{\mathcal{T}}^{\text{val}}$ sets. A meta-learner learns about the underlying structure of the task distribution through training on the meta-training set and is evaluated on meta-testing set.

Our work builds upon Model-Agnostic Meta-Learning (MAML) algorithm [77]. MAML seeks an initialization of parameters θ for a meta-learner such that it can be optimized towards a new task with a small number of gradient steps minimizing the task-specific objectives on the training data $\mathcal{D}_{\mathcal{T}}^{\text{train}}$, with

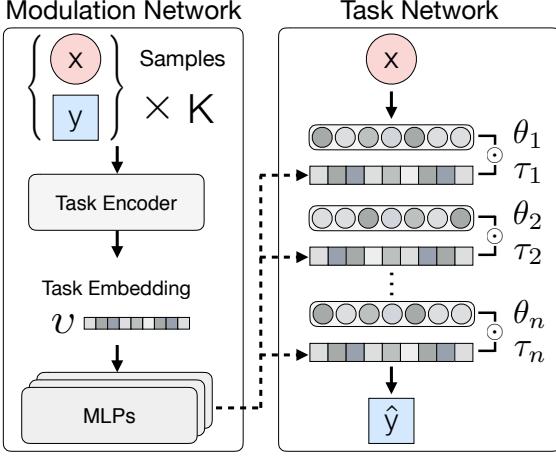


Figure 4.1: Model overview. The modulation network produces a task embedding v , which is used to generate parameters $\{\tau_i\}$ that modulates the task network. The task network adapts modulated parameters to fit to the target task.

Algorithm 1 MMAML META-TRAINING PROCEDURE.

```

1: Input: Task distribution  $P(\mathcal{T})$ , Hyper-parameters  $\alpha$  and  $\beta$ 
2: Randomly initialize  $\theta$  and  $\omega$ .
3: while not DONE do
4:   Sample batches of tasks  $\mathcal{T}_j \sim P(\mathcal{T})$ 
5:   for all  $j$  do
6:     Infer  $v = h(\{x, y\}_K; \omega_h)$  with  $K$  samples from  $\mathcal{D}_{\mathcal{T}_j}^{\text{train}}$ .
7:     Generate parameters  $\tau = \{g_i(v; \omega_g) \mid i = 1, \dots, N\}$  to
       modulate each block of the task network  $f$ .
8:     Evaluate  $\nabla_{\theta} \mathcal{L}_{\mathcal{T}_j}(f(x; \theta, \tau); \mathcal{D}_{\mathcal{T}_j}^{\text{train}})$  w.r.t the  $K$  samples
9:     Compute adapted parameter with gradient descent:
10:     $\theta'_{\mathcal{T}_j} = \theta - \alpha \nabla_{\theta} \mathcal{L}_{\mathcal{T}_j}(f(x; \theta, \tau); \mathcal{D}_{\mathcal{T}_j}^{\text{train}})$ 
11:   end for
12:   Update  $\theta$  with  $\beta \nabla_{\theta} \sum_{T_j \sim P(\mathcal{T})} \mathcal{L}_{\mathcal{T}_j}(f(x; \theta', \tau); \mathcal{D}_{\mathcal{T}_j}^{\text{val}})$ 
13:   Update  $\omega_g$  with  $\beta \nabla_{\omega_g} \sum_{T_j \sim P(\mathcal{T})} \mathcal{L}_{\mathcal{T}_j}(f(x; \theta', \tau); \mathcal{D}_{\mathcal{T}_j}^{\text{val}})$ 
14:   Update  $\omega_h$  with  $\beta \nabla_{\omega_h} \sum_{T_j \sim P(\mathcal{T})} \mathcal{L}_{\mathcal{T}_j}(f(x; \theta', \tau); \mathcal{D}_{\mathcal{T}_j}^{\text{val}})$ 
15: end while

```

the adapted parameters generalize well to the validation data $\mathcal{D}_{\mathcal{T}}^{\text{val}}$. The initialization of the parameters is trained by sampling mini-batches of tasks from \mathcal{D} , computing the adapted parameters for all $\mathcal{D}_{\mathcal{T}}^{\text{train}}$ in the batch, evaluating adapted parameters to compute the validation losses on the $\mathcal{D}_{\mathcal{T}}^{\text{val}}$ and finally update the initial parameters θ using the gradients from the validation losses.

4.4 Method

Our goal is to develop a framework to quickly master a novel task from *a multimodal task distribution*. We call the proposed framework Multimodal Model-Agnostic Meta-Learning (MMAML). The main idea of MMAML is to leverage two complementary neural networks to quickly adapt to a novel task. First, a network called the modulation network predicts the identity of the mode of a task. Then the predicted mode identity is used as an input by a second network called the task network, which is further adapted to the task using gradient-based optimization. Specifically, the modulation network accesses data points from the target task and produces a set of task-specific parameters to modulate the meta-learned prior parameters of the task network. Finally, the modulated task network (but not the task-specific parameters from modulation

network) is further adapted to target task through gradient-based optimization. A conceptual illustration can be found in Figure 4.1.

In the rest of this section, we introduce our modulation network and a variety of modulation operators in Section 4.4.1. Then we describe our task network and the training details for MMAML in Section 4.4.2.

4.4.1 Modulation Network

As mentioned above, modulation network is responsible for identifying the mode of a sampled task, and generate a set of parameters specific to the task. To achieve this, it first takes the given K data points and their labels $\{(x_k, y_k)\}_{k=1,\dots,K}$ as input to the task encoder f and produces an embedding vector v that encodes the characteristics of a task:

$$v = h\left(\{(x_k, y_k) \mid k = 1, \dots, K\}; \omega_h\right) \quad (4.1)$$

Then the task-specific parameters τ are computed based on the encoded task embedding vector v , which is further used to modulate the meta-learned prior parameters of the task network. The task network (introduced later in Section 4.4.2) can be an arbitrarily parameterized function, with multiple building blocks (or layers) such as deep convolutional networks [110], or multi-layer recurrent networks [234]. To modulate the parameters of each block in the task network as good initialization for solving the target task, we apply block-wise transformations to scale and shift the output activation of each hidden unit in the network (*i.e.* the output of a channel of a convolutional layer or a neuron of a fully-connected layer). Specifically, the modulation network produces the modulation vectors for each block i , denoted as

$$\tau_i = g_i(v; \omega_g), \text{ where } i = 1, \dots, N, \quad (4.2)$$

where N is the number of blocks in the task network. We formalize the procedure of applying modulation as:

$\phi_i = \theta_i \odot \tau_i$, where ϕ_i is the modulated prior parameters for the task network, and \odot represents a general modulation operator. We investigate some representative modulation operations including attention-based (softmax) modulation [199, 305] and feature-wise linear modulation (FiLM) [231, 220, 121]. We empirically observe that FiLM performs better and more stable than attention-based modulation (see Section 4.5 for details), and therefore use FiLM as default operator for modulation. The details of these modulation operators can be found in Appendix (Section 4.7).

4.4.2 Task Network

The parameters of each block of the task network are modulated using the task-specific parameters $\tau = \{\tau_i \mid i = 1, \dots, N\}$ generated by the modulation network, which can generate a mode-aware initialization in the parameter space $f(x; \theta, \tau)$. After the modulation step, few steps of gradient descent are performed on the meta-learned prior parameters of the task network to further optimize the objective function for a target task \mathcal{T}_i . Note that the task-specific parameters τ_i are kept fixed and only the meta-learned prior parameters of the task network are updated. We describe the concrete procedure in the form of the pseudo-code as shown in Algorithm 1. The same procedure of modulation and gradient-based optimization is used both during meta-training and meta-testing time.

Detailed network architectures and training hyper-parameters are different by the domain of applications, we defer the complete details to Appendix.

4.5 Experiments

We evaluate our method (MMAML) and baselines in a variety of domains including regression, image classification, and reinforcement learning, under the multimodal task distributions. We consider the following model-agnostic meta-learning baselines:

Table 4.1: Mean square error (MSE) on the **multimodal 5-shot regression** with 2, 3, and 5 modes. A Gaussian noise with $\mu = 0$ and $\sigma = 0.3$ is applied. Multi-MAML uses ground-truth task modes to select the corresponding MAML model. Our method (with FiLM modulation) outperforms other methods by a margin.

Method	2 Modes		3 Modes		5 Modes	
	Post Modulation	Post Adaptation	Post Modulation	Post Adaptation	Post Modulation	Post Adaptation
MAML [77]	-	1.085	-	1.231	-	1.668
Multi-MAML	-	0.433	-	0.713	-	1.082
LSTM Learner	0.362	-	0.548	-	0.898	-
Ours: MMAML (Softmax)	1.548	0.361	2.213	0.444	2.421	0.939
Ours: MMAML (FiLM)	2.421	0.336	1.923	0.444	2.166	0.868

- **MAML** [77] represents the family of model-agnostic meta-learners. The architecture of MAML on each individual domain is designed to be the same as task network in MMAML.
- **Multi-MAML** consists of M (the number of modes) MAML models and each of them is specifically trained on the tasks sampled from a single mode. The performance of this baseline is evaluated by choosing models based on *ground-truth task-mode labels*. This baseline can be viewed as the upper-bound of performance for MAML. If it outperforms MAML, it indicates that MAML’s performance is degenerated due to the multimodality of task distributions. Note that directly comparing the other algorithms to Multi-MAML is not fair as it uses additional information which is not available in real world scenarios.

Note that we aim to develop a general model-agnostic meta-learning framework and therefore the comparison to methods that achieved great performance on only an individual domain are omitted. A more detailed discussion can be found in Appendix (Section 4.7).

4.5.1 Regression Experiments

Setups. We experiment with our models in multimodal few-shot regression. In our setup, five pairs of input/output data $\{x_k, y_k\}_{k=1,\dots,K}$ are sampled from a one dimensional function and provided to a learning

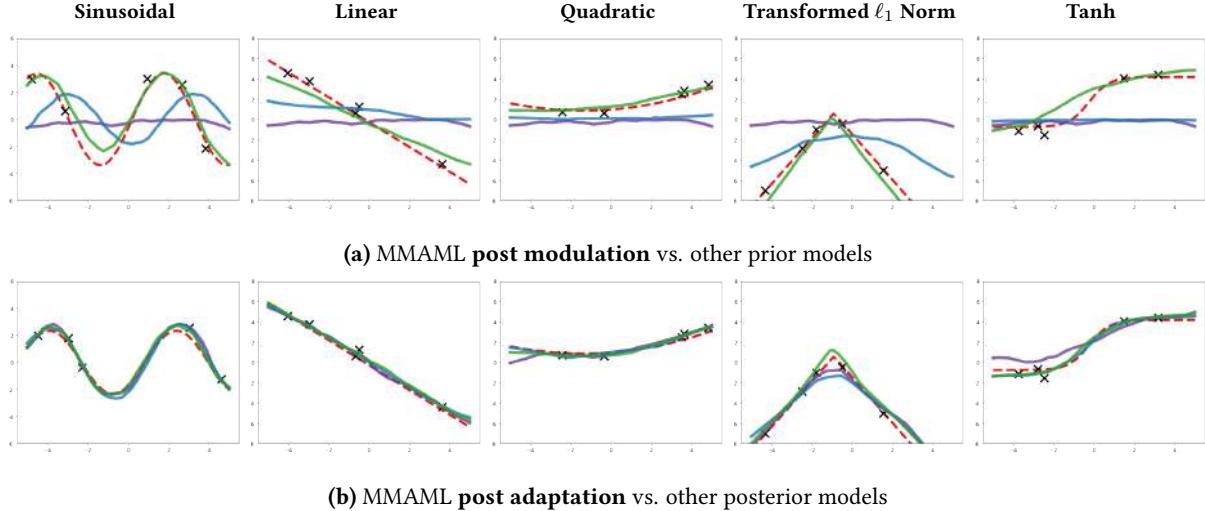


Figure 4.2: Qualitative Visualization of Regression on Five-modes Simple Functions Dataset. **(a)**: We compare the predicted function shapes of modulated MMAML against the prior models of MAML and Multi-MAML, before gradient updates. Our model can fit the target function with limited observations and no gradient updates. **(b)**: The predicted function shapes after five steps of gradient updates, MMAML is qualitatively better. More visualizations in Appendix (Section 4.7).

model. The model is asked to predict L output values y_1^q, \dots, y_L^q for input queries x_1^q, \dots, x_L^q . To construct the multimodal task distribution, we set up five different functions: sinusoidal, linear, quadratic, transformed ℓ_1 norm, and hyperbolic tangent functions, and treat them as discrete task modes. We then evaluate three different task combinations with two functions, three functions and five functions in them. For each task, five pairs of data are sampled and Gaussian noise is added to the output value y , which further increases the difficulty of identifying which function generated the data. Please refer to Appendix (Section 4.7) for details and parameters for regression experiments.

Baselines and Our Approach. As mentioned before, we have MAML and Multi-MAML as two baseline methods, both with MLP task networks. Our method (MMAML) augments the task network with a modulation network. We choose to use an LSTM to serve as the modulation network due to its nature as good at handling sequential inputs and generate predictive outputs. Data points (sorted by x value) are first input to this network to generate task-specific parameters that modulate the task network. The

modulated task network is then further adapted using gradient-based optimization. Two variants of modulation operators – softmax and FiLM are explored to be used in our approach. Additionally, to study the effectiveness of the LSTM model, we evaluate another baseline (referred to as the LSTM Learner) that uses the LSTM as the modulation network (with FiLM) but does not perform gradient-based updates. Please refer to Appendix Section 4.7 for concrete specification of each model.

Results. The quantitative results are shown in Table 4.1. We observe that MAML has the highest error in all settings and that incorporating task identity (Multi-MAML) can improve over MAML significantly. This suggests that MAML degenerates under multimodal task distributions. The LSTM learner outperforms both MAML and Multi-MAML, showing that the sequence model can effectively tackle this regression task. MMAML improves over the LSTM learner significantly, which indicates that with a better initialization (produced by the modulation network), gradient-based optimization can lead to superior performance. Finally, since FiLM outperforms Softmax consistently in the regression experiments, we use it for as the modulation method in the rest of experiments.

We visualize the predicted function shapes of MAML, Multi-MAML and MMAML (with FiLM) in Figure 4.2. We observe that modulation can significantly modify the prediction of the initial network to be close to the target function (see Figure 4.2 (a)). The prediction is then further improved by gradient-based optimization (see Figure 4.2 (b)). tSNE [180] visualization of the task embedding (Figure 4.3) shows that our embedding learns to separate the input data of different tasks, which can be seen as a evidence of the mode identification capability of MMAML.

4.5.2 Image Classification

Setup. The task of few-shot image classification considers the problem of classifying images into N classes with a small number (K) of labeled samples available (*i.e.* N -way K -shot). To create a multimodal few-shot image classification task, we combine multiple widely used datasets (OMNIGLOT [151], MINI-IMAGENET [244],

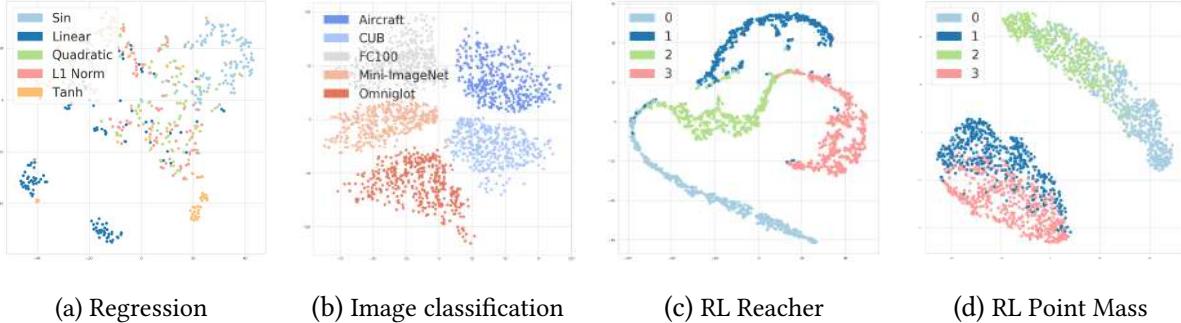


Figure 4.3: tSNE plots of the task embeddings produced by our model from randomly sampled tasks; marker color indicates different modes of a task distribution. The plots (b) and (d) reveal a clear clustering according to different task modes, which demonstrates that MMAML is able to identify the task from a few samples and produce a meaningful embedding v . (a) Regression: the distance between modes aligns with the intuition of the similarity of functions (e.g. a quadratic function can sometimes be similar to a sinusoidal or a linear function while a sinusoidal function is usually different from a linear function) (b) Few-shot image classification: each dataset (*i.e.* mode) forms its own cluster. (c-d) Reinforcement learning: The numbered clusters represent different modes of the task distribution. The tasks from different modes are clearly clustered together in the embedding space.

Table 4.2: Classification testing accuracies on the **multimodal few-shot image classification** with 2, 3, and 5 modes. Multi-MAML uses ground-truth dataset labels to select corresponding MAML models. Our method outperforms MAML and achieve comparable results with Multi-MAML in all the scenarios.

Method & Setup	2 Modes			3 Modes			5 Modes		
	5-way	20-way	5-way	20-way	5-way	20-way	5-way	20-way	5-way
Way	1-shot	5-shot	1-shot	1-shot	5-shot	1-shot	1-shot	5-shot	1-shot
Shot	66.80%	77.79%	44.69%	54.55%	67.97%	28.22%	44.09%	54.41%	28.85%
MAML [77]	66.85%	73.07%	53.15%	55.90%	62.20%	39.77%	45.46%	55.92%	33.78%
MMAML (ours)	69.93%	78.73%	47.80%	57.47%	70.15%	36.27%	49.06%	60.83%	33.97%

FC100 [217], CUB [316], and AIRCRAFT [181]) to form a meta-dataset following the train/test splits used in the prior work, similar to [302]. The details of all the datasets can be found in Appendix (Section 4.7).

We train models on the meta-datasets with two modes (OMNIGLOT and MINI-IMAGENET), three modes (OMNIGLOT, MINI-IMAGENET, and FC100), and five modes (all the five datasets). We use a 4-layer convolutional network for both MAML and our task network.

Results. The results are shown in Table 4.2, demonstrating that our method achieves better results against MAML and performs comparably to Multi-MAML. The performance gap between ours and MAML becomes larger when the number of modes increases, suggesting our method can handle multimodal task

distributions better than MAML. Also, compared to Multi-MAML, our method achieves slightly better results partially because our method learns from all the datasets yet each Multi-MAML is likely to overfit to a single dataset with a smaller number of classes (e.g. MINI-IMAGENET and FC100). This finding aligns with the current trend of meta-learning from multiple datasets [302]. The detailed performance on each dataset can be found in Appendix (Section 4.7).

To gain insights to the task embeddings v produced by our model, we randomly sample 2000 5-mode 5-way 1-shot tasks and employ tSNE to visualize v in Figure 4.3 (b), showing that our task embedding network captures the relationship among modes, where each dataset forms an individual cluster. This structure shows that our task encoder learns a reasonable task embedding space, which allows the modulation network to modulate the parameters of the task network accordingly.

4.5.3 Reinforcement Learning

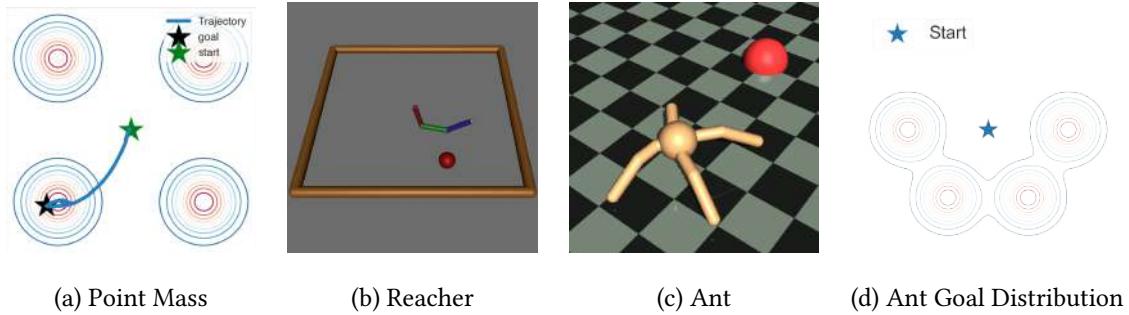


Figure 4.4: RL environments. Three environments are used to explore the capability of MMAML to adapt in multimodal task distributions in RL. In all of the environments the agent is tasked to reach a goal marked by a star or a sphere in the figures. The goals are sampled from a multimodal distribution in two or three dimensions depending on the environment. In POINT MASS (a) the agent navigates a simple point mass agent in 2-dimensions. In REACHER (b) the agent controls a 3-link robot arm in 2-dimensions. In ANT (c) the agent controls four-legged ant robot and has to navigate to the goal. The goals are sampled from a 2-dimensional distribution presented in figure (d), while the agent itself is 3-dimensional.

Setup. Along with few-shot classification and regression, reinforcement learning (RL) has been a central problem where meta-learning has been studied [263, 262, 318, 77, 193, 252]. Similarly to the other domains, the objective in meta-reinforcement learning (meta-RL) is to adapt to a novel task based on limited experience with the task. For RL problems, the inner loop updates of gradient-based meta-learning take the form of

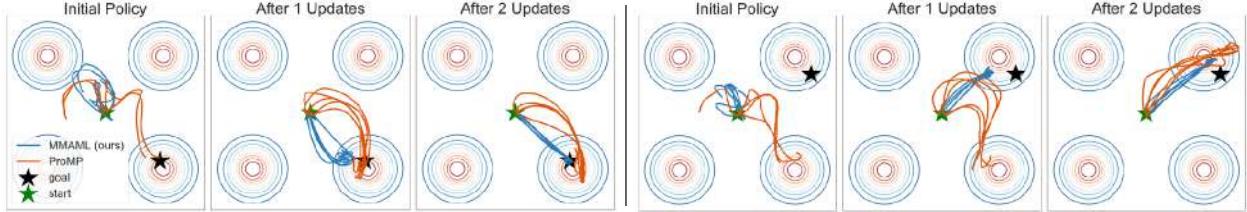


Figure 4.5: Visualizations of MMAML and ProMP trajectories in the 4-mode Point Mass 2D environment. Each trajectory originates in the green star. The contours present the multimodal goal distribution. Multiple trajectories are shown per each update step. For each column: **the leftmost figure** depicts the initial exploratory trajectories without modulation or gradient adaptation applied. **The middle figure** presents ProMP after one gradient adaptation step and MMAML after a gradient adaptation step and the modulation step, which are computed based on the same initial trajectories. **The figure on the right** presents the methods after two gradient adaptation steps in addition to the MMAML modulation step.

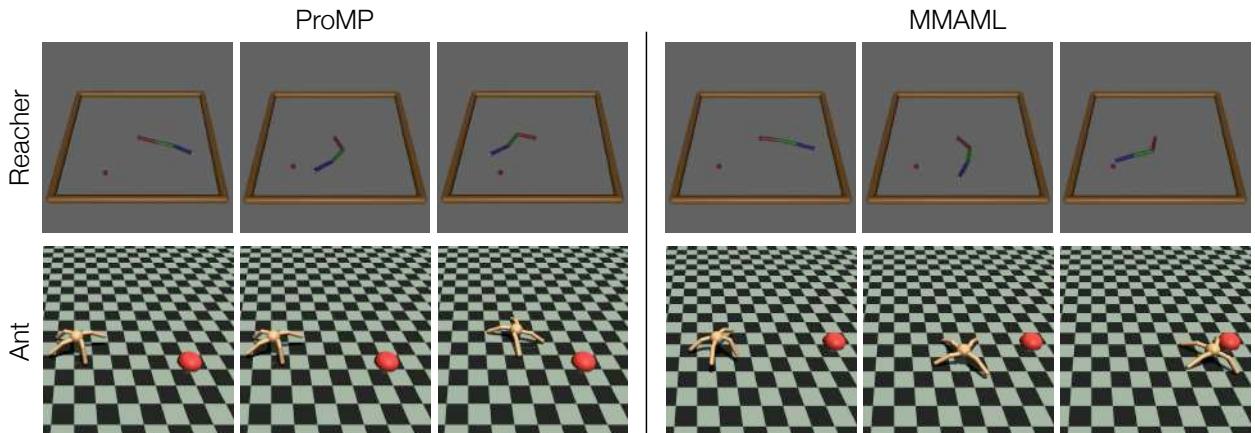


Figure 4.6: Visualizations of MMAML and ProMP trajectories in the ANT and REACHER environments. The figures represent randomly sampled trajectories after the modulation step and two gradient steps for REACHER and three for ANT. Each frame sequence represents a complete trajectory, with the beginning, middle and end of the trajectories captured by the left, middle and right frames respectively. Videos of the trained agents can be found at <https://vuoristo.github.io/MMAML/>.

policy gradient updates. For a more detailed description of the meta-RL problem setting, we refer the reader to [252].

We seek to verify the ability of MMAML to learn to adapt to tasks sampled from multimodal task distributions based on limited interaction with the environment. We do so by evaluating MMAML and the baselines on four continuous control environments using the MuJoCo physics simulator [299]. In all of the environments, the agent is rewarded on every time step for minimizing the distance to the goal. The goals are sampled from multimodal goal distributions with environment specific parameters. The agent does not observe the location of the goal directly but has to learn to find it based on the reward instead. To provide intuition on the environments, illustrations of the robots are presented in Figure 4.4. Examples of trajectories are presented in Figure 4.5 for POINT MASS and in Figure 4.6 for ANT and REACHER. Complete details of the environments and goal distributions can be found in Appendix (Section 4.7).

Baselines and Our Approach. To identify the mode of a task distribution with MMAML, we run the initial policy to interact with the environment and collect a batch of trajectories. These initial trajectories are used for two purposes: computing the adapted parameters using a gradient-based update and modulating the updated parameters based on the task embedding v computed by the modulation network. The modulation vectors τ are kept fixed for the subsequent gradient updates. Descriptions of the network architectures and training hyperparameters are deferred to Appendix (Section 4.7). Due to credit-assignment problems present in the MAML for RL algorithm [77] as identified in [252], we optimize our policies and modulation networks with ProMP [252] algorithm, which resolves these issues.

We use ProMP both as the training algorithm for MMAML and as a baseline. Multi-ProMP is an artificial baseline to show the performance of training one policy for each mode using ProMP. In practice we train an agent for only one of the modes since the task distributions are symmetric and the agent is initialized to a random pose.

Table 4.3: The mean and standard deviation of cumulative reward per episode for multimodal reinforcement learning problems with 2, 4 and 6 modes reported across 3 random seeds. Multi-ProMP is ProMP trained on an easier task distribution which consists of a single mode of the multimodal distribution to provide an approximate upper limit on the performance on each task.

Method	POINT MASS 2D			REACHER			ANT	
	2 Modes	4 Modes	6 Modes	2 Modes	4 Modes	6 Modes	2 Modes	4 Modes
ProMP [252]	-397 ± 20	-523 ± 51	-330 ± 10	-12 ± 2.0	-13.8 ± 2.5	-14.9 ± 2.9	-761 ± 48	-953 ± 46
Multi-ProMP	-109 ± 6	-109 ± 6	-92 ± 4	-4.3 ± 0.1	-4.3 ± 0.1	-4.3 ± 0.1	-624 ± 38	-611 ± 31
Ours	-136 ± 8	-209 ± 32	-169 ± 48	-10.0 ± 1.0	-11.0 ± 0.8	-10.9 ± 1.1	-711 ± 25	-904 ± 37

Results. The results for the meta-RL experiments presented in Table 4.3 show that MMAML consistently outperforms the unmodulated ProMP. The good performance of Multi-ProMP, which only considers a single mode suggests that the difficulty of adaptation in our environments results mainly from the multiple modes. We find that the difficulty of the RL tasks does not scale directly with the number of modes, *i.e.* the performance gap between MMAML and ProMP for POINT MASS with 6 modes is smaller than the gap between them for 4 modes. We hypothesize the more distinct the different modes of the task distribution are, the more difficult it is for a single policy initialization to master. Therefore, adding intermediate modes (going from 4 to 6 modes) can in some cases make the task distribution easier to learn.

The tSNE visualizations of embeddings of random tasks in the POINT MASS and REACHER domains are presented in Figure 4.3. The clearly clustered embedding space shows that the task encoder is capable of identifying the task mode and the good results MMAML achieves suggest that the modulation network effectively utilizes the task embeddings to tackle the multimodal task distribution.

4.6 Conclusion

We present a novel approach that is able to leverage the strengths of both model-based and model-agnostic meta-learners to discover and exploit the structure of multimodal task distributions. Given a few samples from a target task, our modulation network first identifies the mode of the task distribution and then

modulates the meta-learned prior in a parameter space. Next, the gradient-based meta-learner efficiently adapts to the target task through gradient updates. We empirically observe that our modulation network is capable of effectively recognizing the task modes and producing embeddings that captures the structure of a multimodal task distribution. We evaluated our proposed model in multimodal few-shot regression, image classification and reinforcement learning, and achieved superior generalization performance on tasks sampled from multimodal task distributions.

4.7 Appendix

4.7.1 Details on Modulation Operators

Attention based modulation has been widely used in modern deep learning models and has proved its effectiveness across various tasks [339, 199, 348, 332]. Inspired by the previous works, we employed attention to modulate the prior model. In concrete terms, attention over the outputs of all neurons (Softmax) or a binary gating value (Sigmoid) on each neuron’s output is computed by the modulation network. These modulation vectors τ are then used to scale the pre-activation of each neural network layer \mathbf{F}_θ , such that $\mathbf{F}_\phi = \mathbf{F}_\theta \otimes \tau$. Note that here \otimes represents a channel-wise multiplication.

Feature-wise linear modulation (FiLM) has been proposed to modulate neural networks for achieving the conditioning effects of data from different modalities. We adopt FiLM as an option for modulating our task network parameters. Specifically, the modulation vectors τ are divided into two components τ_γ and τ_β such that for a certain layer of the neural network with its pre-activation \mathbf{F}_θ , we would have $\mathbf{F}_\phi = \mathbf{F}_\theta \otimes \tau_\gamma + \tau_\beta$. It can be viewed as a more generic form of attention mechanism. Please refer to [231] for the complete details. In a recent few-shot image classification paper [217], FiLM modulation is used in a metric learning model and achieves high performance. Similarly, employing FiLM modulation

has been shown effective on a variety of tasks such as image synthesis [136, 220, 121, 6], visual question answering [231, 230], style transfer [71], recognition [119, 329], reading comprehension [65], etc.

4.7.2 Further Discussion on Related Works

Discussions on Task-Specific Adaptation/Modulation. As mentioned in the related work of the main text, some recent works [217, 341, 158] leverage the task-specific adaptation or modulation to achieve few-shot image classification. Now we discuss about them in details. [217] propose to learn a task-specific network that adapts the weight of the visual embedding networks via feature-wise linear modulation (FiLM) [231]. Similarly, [341] learns to perform similar task-specific adaptation for few-shot image classification via Transformer [305]. [158] learns a visual embedding network with a task-specific metric and task-agnostic parameters, where the task-specific metric can be update via a fixed steps of gradient updates similar to [79]. In contrast, we aim to leverage the power of task-specific modulation to develop a more powerful model-agnostic meta-learning framework, which is able to effectively adapt to tasks sampled from a multimodal task distribution. Note that our proposed framework is capable of solving few-shot regression, classification, and reinforcement learning tasks.

4.7.3 Baselines

Since we aim to develop a general model-agnostic meta-learning framework, the comparison to methods that achieved great performance on only an individual domain are omitted.

Image Classification. While Prototypical networks [280], Proto-MAML [302], and TADAM [217] learn a metric space for comparing samples and therefore are not directly applicable to regression and reinforcement learning domains, we believe it would be informative to evaluate those methods on our multimodal image classification setting. For this purpose, we refer the readers to a recent work [302] which presents extensive

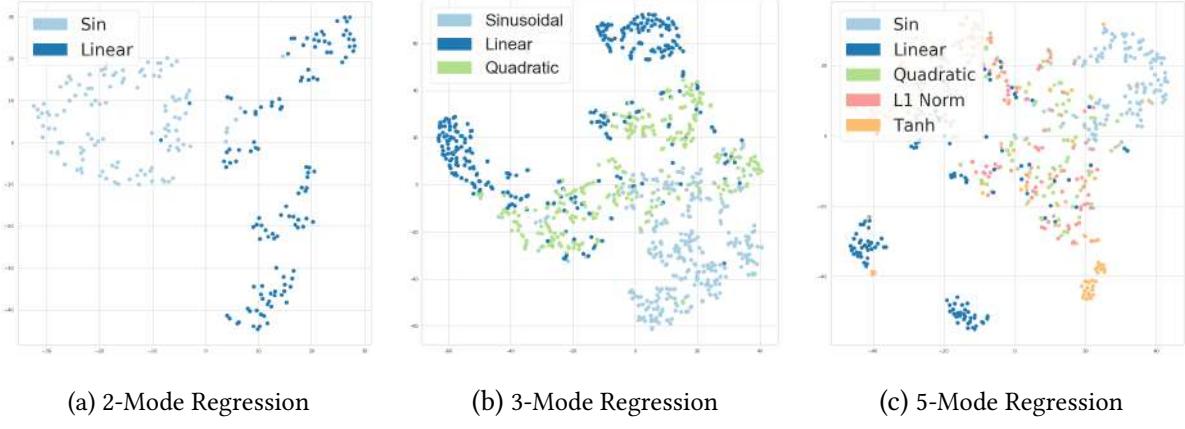


Figure 4.7: tSNE plots of the task embeddings produced by our model from randomly sampled tasks for regression. We choose to visualize the corresponding task embeddings of two modes, three modes and five modes.

experiments on a similar multimodal setting with a wide range of methods, including model-based (RNN-based) methods, model-agnostic meta-learners, and metric-based methods.

Reinforcement Learning. We believe comparing MMAML to ProMP [252] on reinforcement learning tasks highlights the advantage of using a separate modulation network in addition to the task network, given that in the reinforcement learning setting MMAML uses ProMP as the optimization algorithm. Besides ProMP, Bayesian MAML [139] presents an appealing baseline for multimodal task distributions. We tried to run Bayesian MAML on our multimodal task distributions but had technical difficulties with it. The source code for Bayesian MAML in classification and regression is not publicly available.

4.7.4 Additional Experimental Details

4.7.4.1 Regression

Setups

To form multimodal task distributions for regression, we consider a family of functions including sinusoidal functions (in forms of $A \cdot \sin w \cdot x + b + \epsilon$, with $A \in [0.1, 5.0]$, $w \in [0.5, 2.0]$ and $b \in [0, 2\pi]$), linear functions (in forms of $A \cdot x + b$, with $A \in [-3, 3]$ and $b \in [-3, 3]$), quadratic functions (in forms

of $A \cdot (x - c)^2 + b$, with $A \in [-0.15, -0.02] \cup [0.02, 0.15]$, $c \in [-3.0, 3.0]$ and $b \in [-3.0, 3.0]$), ℓ_1 norm function (in forms of $A \cdot |x - c| + b$, with $A \in [-0.15, -0.02] \cup [0.02, 0.15]$, $c \in [-3.0, 3.0]$ and $b \in [-3.0, 3.0]$), and hyperbolic tangent function (in forms of $A \cdot \tanh(x - c) + b$, with $A \in [-3.0, 3.0]$, $c \in [-3.0, 3.0]$ and $b \in [-3.0, 3.0]$). Gaussian observation noise with $\mu = 0$ and $\epsilon = 0.3$ is added to each data point sampled from the target task. In all the experiments, K is set to 5 and L is set to 10. We report the mean squared error (MSE) as the evaluation criterion. Due to the multimodality and uncertainty, this setting is more challenging comparing to [77].

Models and Optimization

In the regression task, we trained a 4-layer fully connected neural network with the hidden dimensions of 100 and ReLU non-linearity for each layer, as the base model for both MAML and MMAML. In MMAML, an additional model with a Bidirectional LSTM of hidden size 40 is trained to generate τ and to modulate each layer of the base model. We used the same hyper-parameter settings as the regression experiments presented in [77] and used Adam [140] as the meta-optimizer. For all our models, we train on 5 meta-train examples and evaluate on 10 meta-val examples to compute the loss.

Evaluation Protocol

In the evaluation of regression experiments, we samples 25,000 tasks for each task mode and evaluate all models with 5 gradient steps during the adaptation (if applicable), with the adaptation learning rate set to be the one models learned with. Therefore, the results for 2 mode experiments is computed over 50,000 tasks, corresponding 3 mode experiment is computed over 75,000 tasks and 5 mode has 125,000 tasks in total. We evaluate all methods over the function range between -5 and 5, and report the accumulated mean squared error as performance measures.

Effect of Modulation and Adaptation

We analyze the effect of modulation and adaptation steps on the regression experiments. Specifically, we show both the qualitative and quantitative results on the 5-mode regression task, and plot the induced

function curves as well as measure the Mean Squared Error (MSE) after applying modulation step or both modulation and adaptation step. Note that MMAML starts from a learned prior parameters (denoted as *prior params*), and then sequentially performs modulation and adaptation steps. The results are shown in the Figure 4.8 and Table 4.4. We see that while inference with prior parameters itself induces high error, adding modulation as well as further adaptation can significantly reduce such error. We can see that the modulation step is trying to seek a rough solution that captures the shape of the target curve, and the gradient based adaptation step refines the induced curve.

Figure 4.8: 5-mode Regression: Visualization with Linear & Quadratic Function.

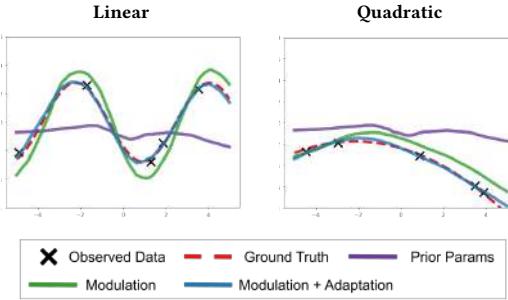


Table 4.4: 5-mode Regression: Performance measured in mean squared error (MSE).

MMAML	MSE
Prior Params	17.299
+ Modulation	2.166
+ Adaptation	0.868

4.7.4.2 Image Classification

Meta-dataset

To create a meta-dataset by merging multiple datasets, we utilize five popular datasets: OMNIGLOT, MINI-IMAGENET, FC100, CUB, and AIRCRAFT. The detailed information of all the datasets are summarized in Table 4.5. To fit the images from all the datasets to a model, we resize all the images to 84×84 . The images randomly sampled from all the datasets are shown in Figure 4.9, demonstrating a diverse set of modes.

Hyperparameters

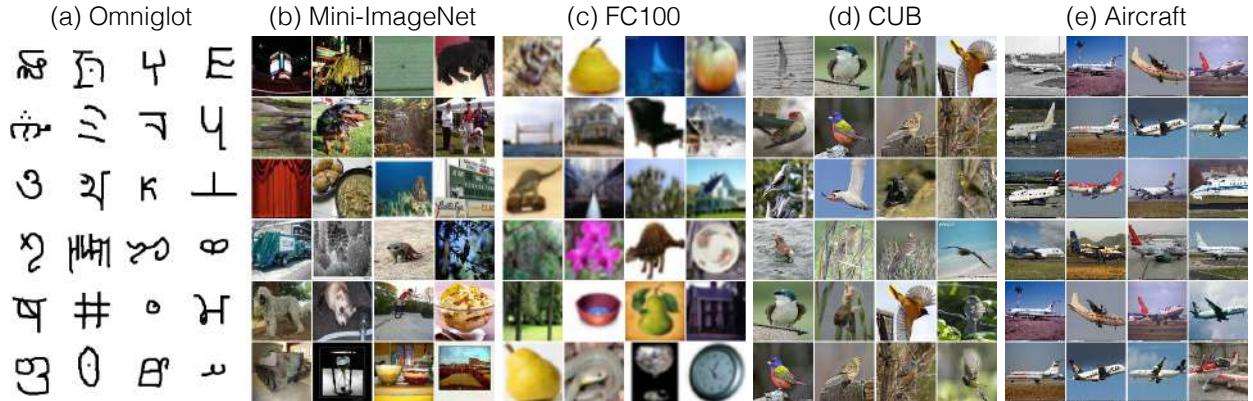


Figure 4.9: Examples of images from all the datasets.

Table 4.5: Details of few-shot image classification datasets.

Dataset	Train classes	Validation classes	Test classes	Image size	Image channel	Image content
OMNIGLOT	4112	688	1692	28×28	1	handwritten characters
MINI-IMAGESET	64	16	20	84×84	3	objects
FC100	64	16	20	32×32	3	objects
CUB	140	30	30	$\sim 500 \times 500$	3	birds
AIRCRAFT	70	15	15	$\sim 1\text{-}2$ Mpixels	3	aircrafts

We present the hyperparameters for all the experiments in Table 4.6. We use the same set of hyperparameters to train our model and MAML for all experiments, except that we use a smaller meta batch-size for 20-way tasks and train the jobs for more iterations due to the limited memory of GPUs that we have access to.

We use 15 examples per class for evaluating the post-update meta-gradient for all the experiments, following [77, 244]. All the trainings use the Adam optimizer [140] with default hyperparameters.

For Multi-MAML, since we train a MAML model for each dataset, it gives us the freedom to use different sets of hyperparameters for different datasets. We tried our best to find the best hyperparameters for each dataset.

Network Architectures

Table 4.6: Hyperparameters for multimodal few-shot image classification experiments. We experiment different hyperparameters for each dataset for Multi-MAML. The dataset group **Grayscale** includes OMNIGLOT and **RGB** includes MINI-IMAGENET and FC100, CUB, and AIRCRAFT.

Method	Setup	Dataset group	Slow lr	Fast lr	Meta batch-size	Number of updates	Training iterations
MAML	5-way 1-shot		-	0.001	10	5	60000
	5-way 5-shot						
	5-way 1-shot						
	5-way 5-shot						
MMAML (ours)	20-way 1-shot		-	0.001	0.05	5	80000
	20-way 3-shot						
	20-way 1-shot						
	20-way 3-shot						
Multi-MAML	5-way 1-shot	Grayscale	0.001	0.4	10	1	60000
		RGB		0.01	4	5	
	5-way 5-shot	Grayscale		0.4	10	1	
		RGB		0.01	4	5	
	20-way 1-shot	Grayscale		0.1	4	5	80000
		RGB		0.01	2	5	
	20-way 3-shot	Grayscale		0.1	4	5	
		RGB		0.01	2	5	

Task Network. For the task network, we use the exactly same architecture as the MAML convolutional network proposed in [77]. It consists of four convolutional layers with the channel size 32, 64, 128, and 256, respectively. All the convolutional layers have a kernel size of 3 and stride of 2. A batch normalization layer follows each convolutional layer, followed by ReLU. With the input tensor size of $(n \cdot k) \times 84 \times 84 \times 3$ for a n -way k -shot task, the output feature maps after the final convolutional layer have a size of $(n \cdot k) \times 6 \times 6 \times 256$. The feature maps are then average pooled along spatial dimensions, resulting feature vectors with a size of $(n \cdot k) \times 256$. A linear fully-connected layer takes the feature vector as input, and produce a classification prediction with a size of n for n-way classification tasks.

Task Encoder. For the task encoder, we use the exactly same architecture as the task network. It consists of four convolutional layers with the channel size 32, 64, 128, and 256, respectively. All the convolutional layers have a kernel size of 3, stride of 2, and use valid padding. A batch normalization layer follows each convolutional layer, followed by ReLU. With the input tensor size of $(n \cdot k) \times 84 \times 84 \times 3$ for a n -way

Table 4.7: The performance (classification accuracy) on the **multimodal few-shot image classification** with **2 modes** on each dataset.

Setup	Method	Datasets		
		OMNIGLOT	MINI-IMAGENET	OVERALL
5-way 1-shot	MAML	89.24%	44.36%	66.80%
	Multi-MAML	97.78%	35.91%	66.85%
	MMAML (ours)	94.90%	44.95%	69.93%
5-way 5-shot	MAML	96.24%	59.35%	77.79%
	Multi-MAML	98.48%	47.67%	73.07%
	MMAML (ours)	98.47%	59.00%	78.73%
20-way 1-shot	MAML	55.36%	15.67%	35.52%
	Multi-MAML	91.59%	14.71%	53.15%
	MMAML (ours)	83.14%	12.47%	47.80%

Table 4.8: The performance (classification accuracy) on the **multimodal few-shot image classification** with **3 modes** on each dataset.

Setup	Method	Datasets			
		OMNIGLOT	MINI-IMAGENET	FC100	OVERALL
5-way 1-shot	MAML	86.76%	43.27%	33.29%	54.55%
	Multi-MAML	97.78%	35.91%	34.00%	55.90%
	MMAML (ours)	93.67%	41.07%	33.67%	57.47%
5-way 5-shot	MAML	95.11%	61.48%	47.33%	67.97%
	Multi-MAML	98.48%	47.67%	40.44%	62.20%
	MMAML (ours)	99.56%	60.67%	50.22%	70.15%
20-way 1-shot	MAML	57.87%	15.06%	11.74%	28.22%
	Multi-MAML	91.59%	14.71%	13.00%	39.77%
	MMAML (ours)	85.00%	13.00%	10.81%	36.27%

k -shot task, the output feature maps after the final convolutional layer have a size of $(n \cdot k) \times 6 \times 6 \times 256$.

The feature maps are then average pooled along spatial dimensions, resulting feature vectors with a size of $(n \cdot k) \times 256$. To produce an aggregated embedding vector from all the feature vectors representing all samples, we perform an average pooling, resulting a feature vector with a size of 256. Finally, a fully-connected layer followed by ReLU takes the feature vector as input, and produce a task embedding vector v with a size of 128.

Table 4.9: The performance (classification accuracy) on the **multimodal few-shot image classification** with **5 modes** on each dataset.

Setup	Method	Datasets					
		OMNIGLOT	MINI-IMAGE NET	FC100	CUB	AIRCRAFT	OVERALL
5-way 1-shot	MAML	83.63%	37.78%	33.70%	86.96%	36.74%	35.48%
	Multi-MAML	97.78%	35.91%	34.00%	93.44%	32.03%	27.59%
	MMAML (ours)	91.48%	42.89%	32.59%	93.56%	38.30%	36.82%
5-way 5-shot	MAML	89.41%	51.26%	43.41%	82.30%	45.80%	43.92%
	Multi-MAML	98.48%	47.67%	40.44%	98.56%	45.70%	47.29%
	MMAML (ours)	97.96%	51.29%	44.08%	97.88%	53.80%	51.53%
20-way 1-shot	MAML	59.10%	15.49%	11.75%	59.45%	16.31%	31.57%
	Multi-MAML	91.59%	14.71%	13.00%	85.46%	18.87%	30.72%
	MMAML (ours)	86.28%	14.35%	11.59%	91.86%	24.05%	30.89%

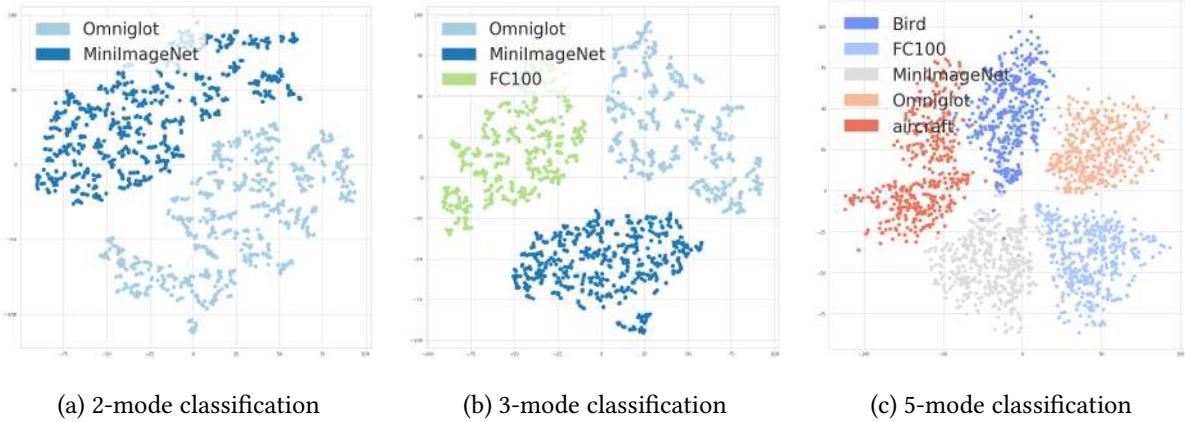


Figure 4.10: tSNE plots of task embeddings produced in multimodal few-shot image classification domain. (a) 2-mode 5-way 1-shot (b) 3-mode 5-way 1-shot (c) 5-mode 5-way 5-shot.

Modulation MLPs. Since the task network consists of four convolutional layers with the channel size 32, 64, 128, and 256 and modulating each of them requires producing both τ_γ and τ_β , we employ four linear fully-connected layers to convert the task embedding vector v to $\{\tau_{\gamma_1}, \tau_{\beta_1}\}$ (with a dimension of 32), $\{\tau_{\gamma_2}, \tau_{\beta_2}\}$ (with a dimension of 64), $\{\tau_{\gamma_3}, \tau_{\beta_3}\}$ (with a dimension of 128), and $\{\tau_{\gamma_4}, \tau_{\beta_4}\}$ (with a dimension of 256). Note the modulation for each layer is performed by $\theta_i \odot \gamma_i + \beta_i$, where \odot denotes the Hadamard product.

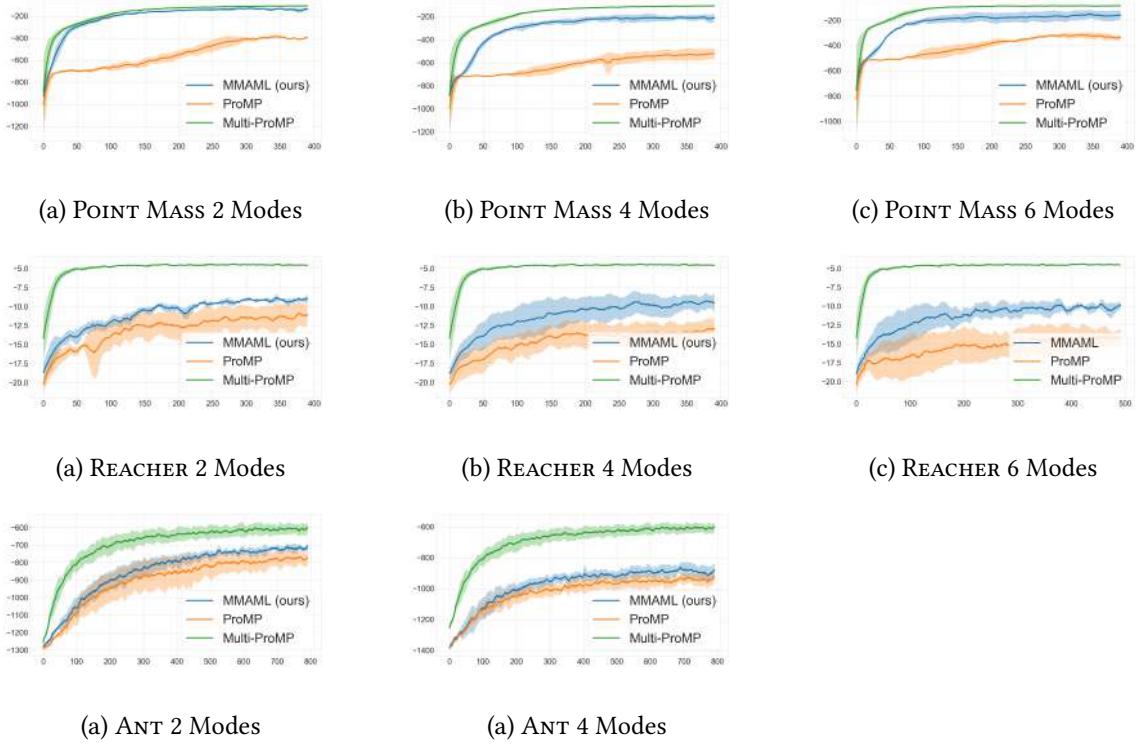


Figure 4.11: Training curves for MMAML and ProMP in reinforcement learning environments. The curves indicate the average return per episode after gradient-based updates and modulation. The shaded region indicates standard deviation across three random seeds. The curves have been smoothed by averaging the values within a window of 10 steps.

4.7.4.3 Reinforcement Learning

Environments

The training curves for all environments are presented in Figure 4.11.

POINT MASS. We consider three variants of the POINT MASS environment with 2, 4, and 6 modes. The agent controls a point mass by outputting changes to the velocity. At every time step the agent receives the negative euclidean distance to the goal as the reward. The goals are sampled from a multimodal goal distribution by first selecting the mode center and then adding Gaussian noise to the goal location. In the 4 mode variant the modes are the points $(-5, -5)$, $(-5, 5)$, $(5, -5)$, $(5, 5)$. In the 2 mode variant the modes are the points $(-5, -5)$, $(5, 5)$. In the 6 mode variant the modes are the vertices of a regular hexagon with

at distance 5 from the origin. All variants have noise scale of 2.0. Visualizations of agent trajectories can be found in Figure 4.13.

REACHER. We consider three variants of the REACHER environment with 2, 4, and 6 modes. The agent controls a 2-dimensional robot arm with three links simulated in the MuJoCo [299] simulator. The goal distribution is similar to the goal distributions in POINT MASS but different parameters are used to match the scale of the environment. The reward for the environment is

$$R(s, a) = -1 * (x_{point} - x_{goal})^2 - \|a\|^2$$

where x_{point} is the location of the point of the arm, x_{goal} if the location of the goal and a is the action chosen by the agent. The modes of the goal distribution in the 4 mode variant are located at $(-0.225, -0.225)$, $(0.225, -0.225)$, $(-0.225, 0.225)$, $(0.225, 0.225)$ and the goal noise has scale of 0.1. In the 2 mode variant the modes are located at $(-0.225, -0.225)$, $(0.225, 0.225)$ and the noise scale is 0.1. In the 6 mode variant the mode centers are the vertices of a regular hexagon with distance to the origin of 0.318 and the noise scale is 0.1.

ANT. We consider two variants of the ANT environment with two and four modes. The agent controls an ant robot with four limbs simulated in the MuJoCo [299] simulator. The reward for the environment is

$$R(s, a) = -1 * (x_{torso} - x_{goal})^2 - \lambda_{control} * \|a\|^2$$

where x_{torso} is the location of the torso of the robot, x_{goal} if the location of the goal, $\lambda_{control} = 0.1$ is the weighting for the control cost and a is the action chosen by the agent. The modes of the goal distribution in the 4 mode variant are located at $(-4, 0)$, $(-2, 3.46)$, $(2, 3.46)$, $(4, 0)$ and the goal noise has scale of 0.8. In the 2 mode variant the modes are located at $(-4, 0)$, $(4, 0)$ and the noise scale is 0.8.

Table 4.10: Hyperparameter settings for reinforcement learning.

Environment	Algorithm	Training Iterations	Trajectory Length	Slow lr	Fast lr	Inner Gradient Steps	Clip eps
MMAML							
POINT MASS	ProMP	400	100	0.0005	0.01	2	0.1
	Multi-ProMP						
MMAML							
REACHER	ProMP	800	50	0.001	0.1	2	0.1
	Multi-ProMP						
MMAML							
ANT	ProMP	800	250	0.001	0.1	3	0.1
	Multi-ProMP						

Network Architectures and Hyperparameters

For all RL experiments we use a policy network with two 64-unit hidden layers. The modulation network in RL tasks consists of a GRU-cell and post processing layers. The inputs to the GRU are the concatenated observations, actions and reward for each trajectory. The trajectories are processed separately. An MLP is used to process the last hidden states of each trajectory. The outputs of the MLPs are averaged and used by another MLP to compute the modulation vectors τ . All MLPs have a single hidden layer of size 64.

We sample 40 tasks for each update step. For each gradient step for each task we sample 20 trajectories. The hyperparameters, which differ from setting to setting are presented in Table 4.10.

4.7.5 Additional Experimental Results

4.7.5.1 Regression

We show visualization of embeddings for regression experiments with a varying number of task modes as Figure 4.7. We observe a linear separation in the two task modes and three task modes scenarios, which indicates that our method is capable of identifying data from different task modes. On the visualization of five task mode, we observe that data from linear, transformed ℓ_1 norm and hyperbolic tangent functions

cluttered. This is due to the fact that those functions are very similar to each other, especially with the Gaussian noise we added in the output space.

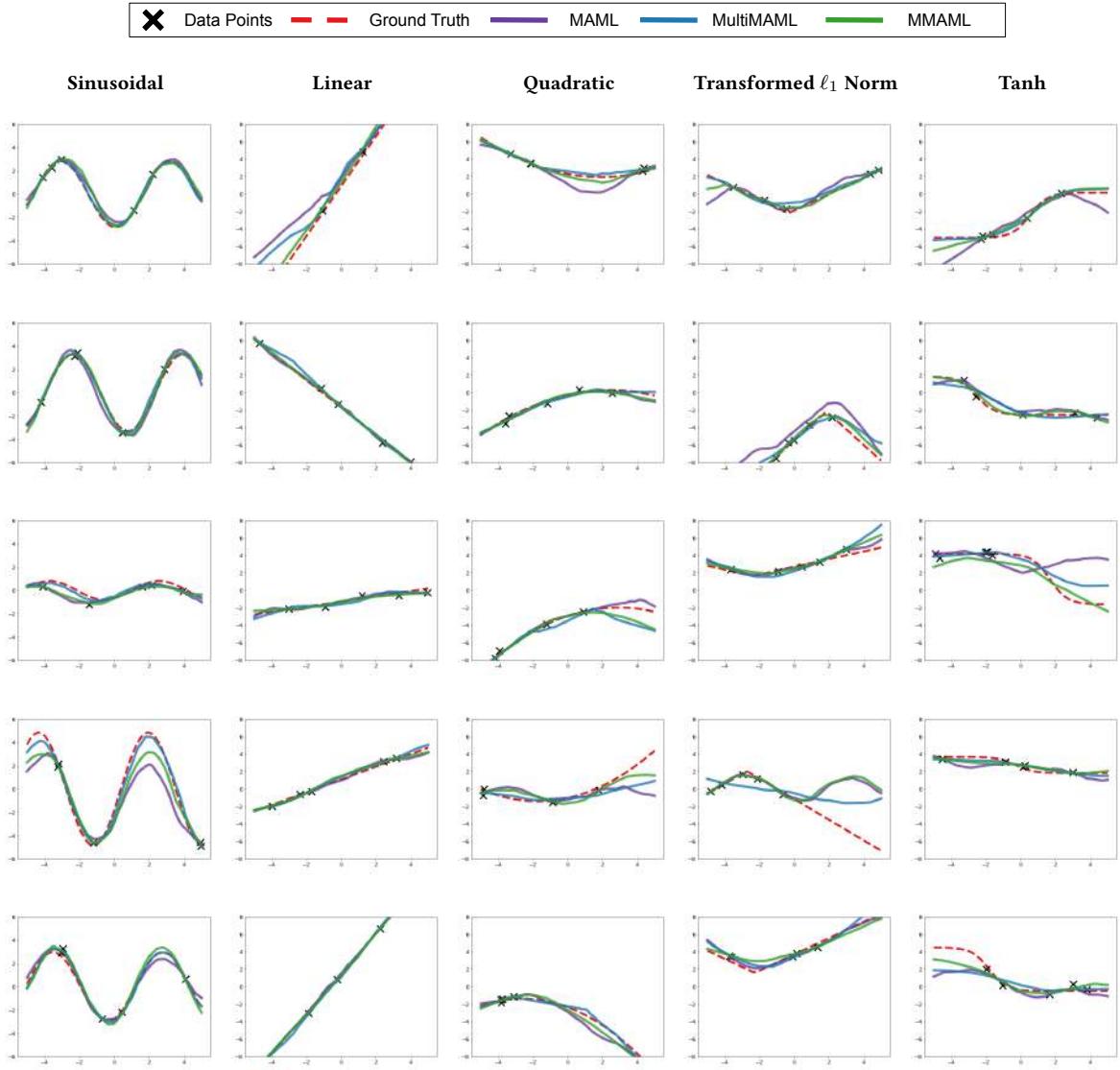


Figure 4.12: Additional qualitative results of the regression tasks. **MMAML after adaptation vs. other posterior models.**

4.7.5.2 Image Classification

We provide the detailed performance of our method and the baselines on each individual dataset for all 2, 3, and 5 mode experiments, shown in Table 4.7, Table 4.8, and Table 4.9, respectively. Note that the main paper presents the overall performance (the last columns of each table) on each of 2, 3, and 5 mode experiments.

We found the results on OMNIGLOT and MINI-IMAGENET demonstrate similar tendency shown in [302]. Note that the performance of OMNIGLOT and FC100 might be slightly different from the results reported in the related papers because (1) all the images are resized and tiled along the spatial dimensions, (2) different hyperparameters are used, and (3) different numbers of training iterations.

Additional tSNE plots for predicted task embeddings of 2-mode 5-way 1-shot classification, 3-mode 5-way 1-shot classification, and 5-mode 20-way 1-shot classification are shown in Figure 4.10.

4.7.5.3 Reinforcement Learning

Additional trajectories sampled from the 2D navigation environment are presented in Figure 4.13.

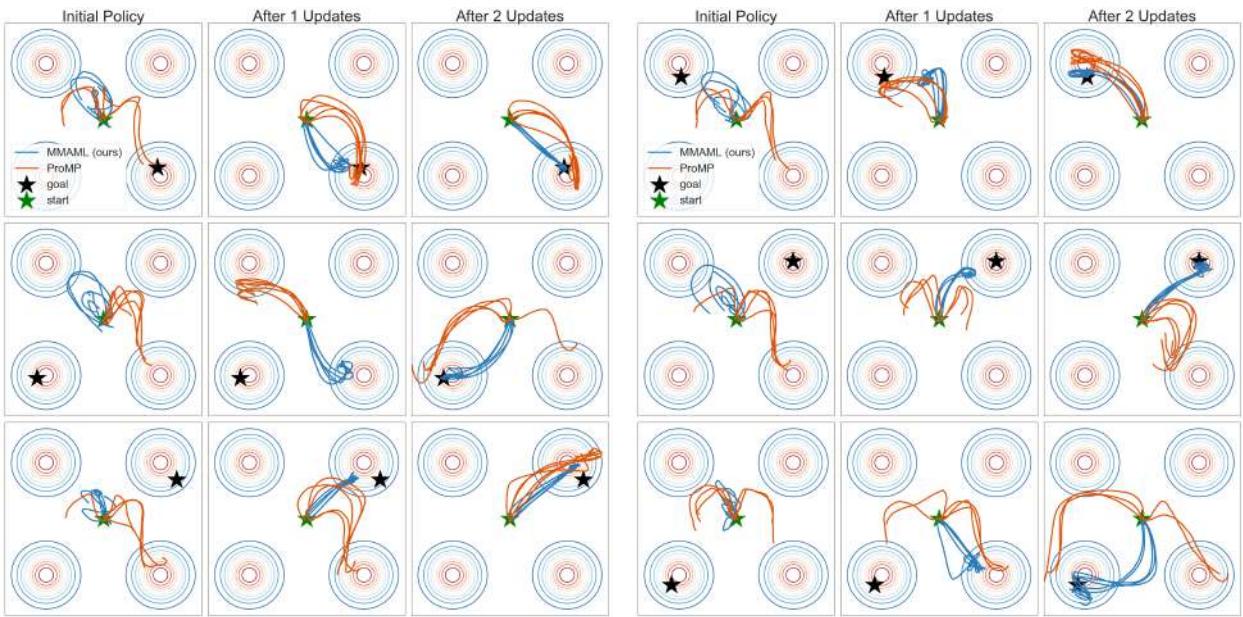


Figure 4.13: Additional trajectories sampled from the point mass environment with MMAML and ProMP for six tasks. The contour plots represents the multimodal task distribution. The stars mark the start and goal locations. The curves depict five trajectories sampled using each method after zero, one and two update steps. In the figure, the modulation step takes place between the initial policy and the step after one update.

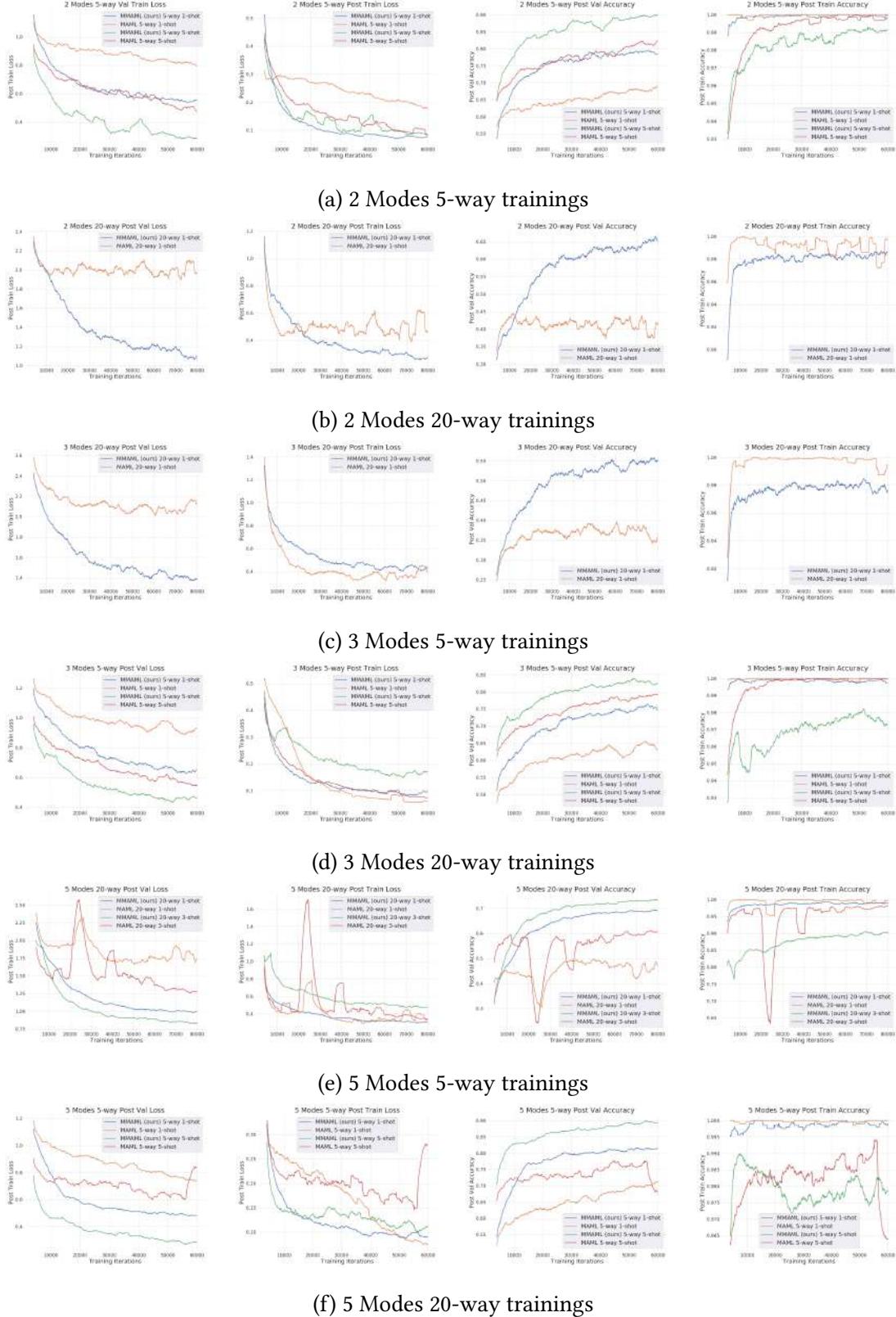


Figure 4.14: Training curves of MAML and our method for few-shot image classification. We show the losses and classification accuracies for training and validation tasks after adaptation. MAML trainings are less stable, while ours curves are smoother.

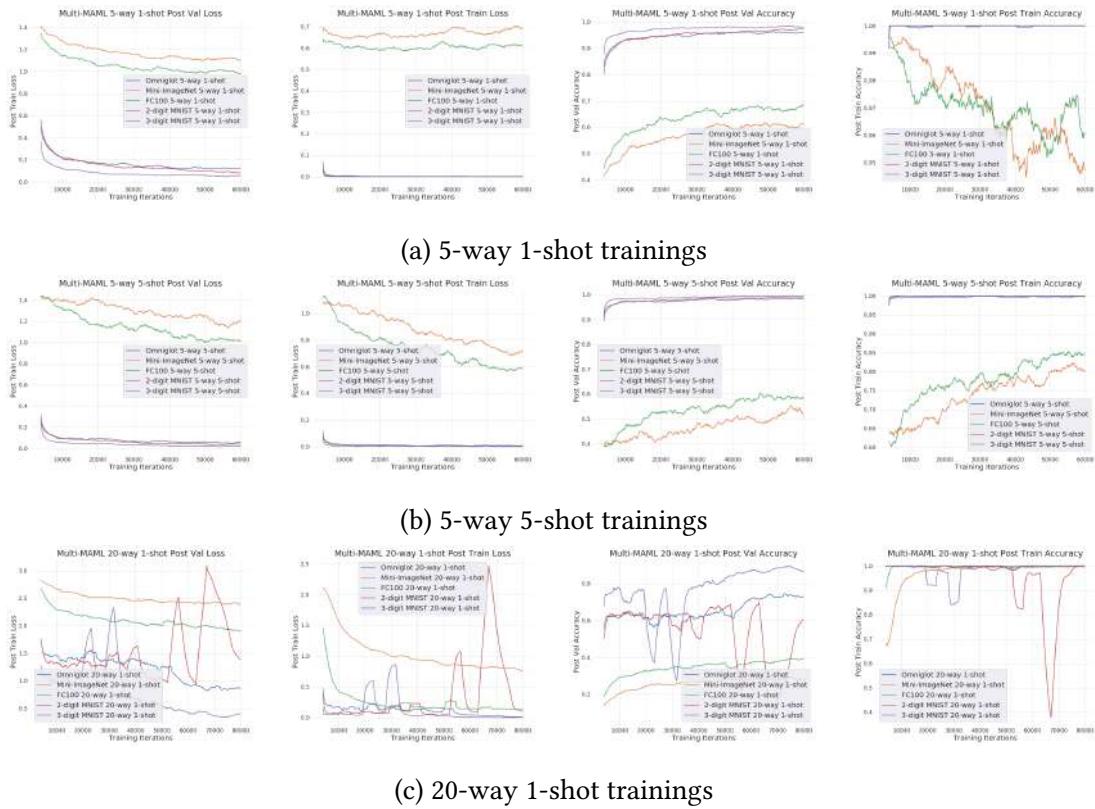


Figure 4.15: Training curves of Multi-MAML for few-shot image classification. We show the losses and classification accuracies for training and validation tasks after adaptation.

Chapter 5

Meta-Learning on Long-Horizon and Sparse-Reward Tasks

5.1 Introduction

In recent years, deep reinforcement learning methods have achieved impressive results in robot learning [98, 15, 134]. Yet, existing approaches are sample inefficient, thus rendering the learning of complex behaviors through trial and error learning infeasible, especially on real robot systems. In contrast, humans are capable of effectively learning a variety of complex skills in only a few trials. This can be greatly attributed to our ability to learn how to learn new tasks quickly by efficiently utilizing previously acquired skills.

Can machines likewise learn to how to learn by efficiently utilizing learned skills like humans? Meta-reinforcement learning (meta-RL) holds the promise of allowing RL agents to acquire novel tasks with improved efficiency by learning to learn from a distribution of tasks [77, 243]. In spite of recent advances in

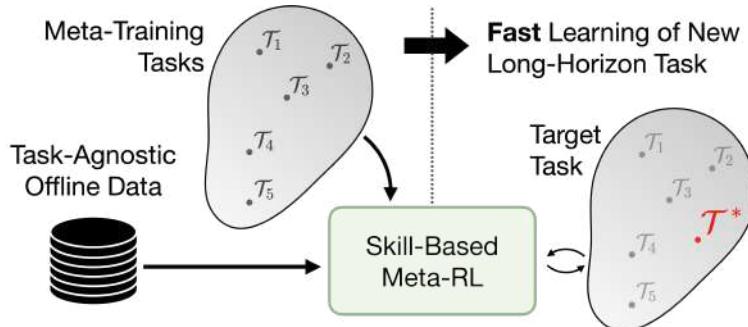


Figure 5.1: **Overview.** We propose a method that jointly leverages (1) a large offline dataset of prior experience collected across many tasks without reward or task annotations and (2) a set of meta-training tasks to learn how to quickly solve unseen long-horizon tasks. Our method extracts reusable skills from the offline dataset and meta-learn a policy to quickly use them for solving new tasks.

the field, most existing meta-RL algorithms are restricted to short-horizon, dense-reward tasks. To facilitate efficient learning on long-horizon, sparse-reward tasks, recent works aim to leverage experience from prior tasks in the form of offline datasets without additional reward and task annotations [179, 232, 42]. While these methods can solve complex tasks with substantially improved sample efficiency over methods learning from scratch, millions of interactions with environments are still required to acquire long-horizon skills.

In this work, we aim to take a step towards combining the capabilities of *both* learning how to quickly learn new tasks while *also* leveraging prior experience in the form of unannotated offline data (see Figure 5.1). Specifically, we aim to devise a method that enables meta-learning on complex, long-horizon tasks and can solve unseen target tasks with orders of magnitude fewer environment interactions than prior works.

We propose to leverage the offline experience by extracting reusable *skills* – short-term behaviors that can be composed to solve unseen long-horizon tasks. We employ a hierarchical meta-learning scheme in which we meta-train a high-level policy to learn how to quickly reuse the extracted skills. To efficiently explore the learned skill space during meta-training, the high-level policy is guided by a skill prior which is also acquired from the offline experience data.

We evaluate our method and prior approaches in deep RL, skill-based RL, meta-RL, and multi-task RL on two challenging continuous control environments: maze navigation and kitchen manipulation, which require long-horizon control and only provides sparse rewards. Experimental results show that our method can efficiently solve unseen tasks by exploiting meta-learning tasks and offline datasets, while prior approaches require substantially more samples or fail to solve the tasks.

In summary, the main contributions of this paper are threefold:

- To the best of our knowledge, this is the first work to combine meta-reinforcement learning algorithms with task-agnostic offline datasets that do not contain reward or task annotations.

- We propose a method that combines meta-learning with offline data by extracting learned skills and a skill prior as well as meta-learning a hierarchical skill policy regularized by the skill prior.
- We empirically show that our method is significantly more efficient at learning long-horizon sparse-reward tasks compared to prior methods in deep RL, skill-based RL, meta-RL, and multi-task RL.

5.2 Related Work

Meta-Reinforcement Learning. Meta-RL approaches [70, 318, 77, 344, 252, 102, 315, 204, 53, 54, 243, 314, 337, 353, 122, 354, 174] hold the promise of allowing learning agents to quickly adapt to novel tasks by learning to learn from a distribution of tasks. Despite the recent advances in the field, most existing meta-RL algorithms are limited to short-horizon and dense-reward tasks. In contrast, we aim to develop a method that can meta-learn to solve long-horizon tasks with sparse rewards by leveraging offline datasets.

Offline datasets. Recently, many works have investigated the usage of offline datasets for agent training. In particular, the field of *offline reinforcement learning* [163, 275, 150, 345] aims to devise methods that can perform RL fully offline from pre-collected data, without the need for environment interactions. However, these methods require target task reward annotations on the offline data for every new tasks that should be learned. These reward annotations can be challenging to obtain, especially if the offline data is collected from a diverse set of prior tasks. In contrast, our method is able to leverage offline datasets without any reward annotations.

Offline Meta-RL. Another recent line of research aims to *meta-learn* from static, pre-collected datasets including reward annotations [196, 238, 68]. After meta-training with the offline datasets, these works aim to quickly adapt to a new task with only a small amount of data from that new task. In contrast to the aforementioned offline RL methods these works aim to adapt to *unseen* tasks and assume access to only *limited data* from the new tasks. However, in addition to reward annotations, these approaches often

require that the offline training data is split into separate datasets for each training tasks, further limiting the scalability.

Skill-based Learning. An alternative approach for leveraging offline data that does not require reward or task annotations is through the extraction of skills – reusable short-horizon behaviors. Methods for skill-based learning recombine these skills for learning unseen target tasks and converge substantially faster than methods that learn from scratch [160, 108, 269]. When trained from diverse datasets these approaches can extract a wide repertoire of skills and learn complex, long-horizon tasks [191, 179, 232, 4, 42, 233]. Yet, although they are more efficient than training from scratch, they still require a large number of environment interactions to learn a new task. Our method instead combines skills extracted from offline data with meta-learning, leading to significantly improved sample efficiency.

5.3 Problem Formulation and Preliminaries

Our approach builds on prior work for meta-learning and learning from offline datasets and aims to combine the best of both worlds. In the following we will formalize our problem setup and briefly summarize relevant prior work.

Problem Formulation. Following prior work on learning from large offline datasets [179, 232, 233], we assume access to a dataset of state-action trajectories $\mathbf{D} = \{s_t, a_t, \dots\}$ which is collected either across a wide variety of tasks or as “play data” with no particular task in mind. We thus refer to this dataset as *task-agnostic*. With a large number of data collection tasks, the dataset covers a wide variety of behaviors and can be used to accelerate learning on diverse tasks. Such data can be collected at scale, e.g. through autonomous exploration [108, 269, 57], human teleoperation [259, 101, 183, 179], or from previously trained agents [87, 100]. We additionally assume access to a set of meta-training tasks $\mathbf{T} = \{\mathcal{T}_1, \dots, \mathcal{T}_N\}$, where each task is represented as a Markov decision process (MDP) defined by a tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \rho, \gamma\}$ of states, actions, transition probability, reward, initial state distribution, and discount factor.

Our goal is to leverage both, the offline dataset \mathbf{D} and the meta-training tasks \mathbf{T} , to accelerate the training of a policy $\pi(a|s)$ on a target task \mathcal{T}^* which is also represented as an MDP. Crucially, we do not assume that \mathcal{T}^* is a part of the set of training tasks \mathbf{T} , nor that \mathbf{D} contains demonstrations for solving \mathcal{T}^* . Thus, we aim to design an algorithm that can leverage offline data and meta-training tasks for learning how to quickly compose known skills for solving an unseen target task. Next, we will describe existing approaches that *either* leverage offline data *or* meta-training tasks to accelerate target task learning. Then, we describe how our approach takes advantage of the best of both worlds.

Skill-based RL. One successful approach for leveraging task-agnostic datasets for accelerating the learning of unseen tasks is through the transfer of reusable *skills*, *i.e.* short-horizon behaviors that can be composed to solve long-horizon tasks. Prior work in skill-based RL called Skill-Prior RL (SPiRL, Pertsch, Lee, and Lim [232]) proposes an effective way to implement this idea. Specifically, SPiRL uses a task-agnostic dataset to learn two models: (1) a skill policy $\pi(a|s, z)$ that decodes a latent skill representation z into a sequence of executable actions and (2) a prior over latent skill variables $p(z|s)$ which can be leveraged to guide exploration in skill space. SPiRL uses these skills for learning new tasks efficiently by training a high-level skill policy $\pi(z|s)$ that acts over the space of learned skills instead of primitive actions. The target task RL objective extends Soft Actor Critic (SAC, Haarnoja et al. [104]), a popular off-policy RL algorithm, by guiding the high-level policy with the learned skill prior:

$$\max_{\pi} \sum_t \mathbb{E}_{(s_t, z_t) \sim \rho_{\pi}} [r(s_t, z_t) - \alpha D_{\text{KL}}(\pi(z|s_t), p(z|s_t))]. \quad (5.1)$$

Here D_{KL} denotes the Kullback-Leibler divergence between the policy and skill prior, and α is a weighting coefficient.

Off-Policy Meta-RL. Rakelly et al. [243] introduced an off-policy meta-RL algorithm called probabilistic embeddings for actor-critic RL (PEARL) that leverages a set of training tasks \mathbf{T} to enable quick learning of

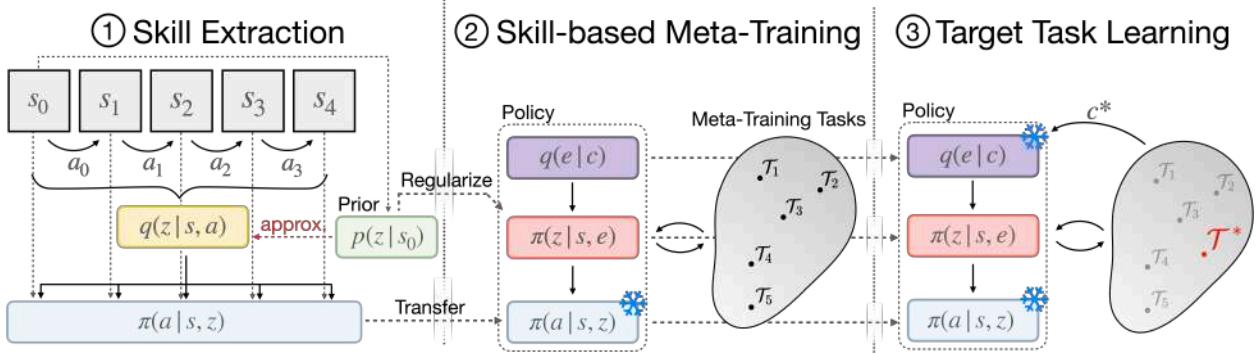


Figure 5.2: **Method Overview.** Our proposed skill-based meta-RL method has three phases. **(1) Skill Extraction:** learns reusable skills from snippets of task-agnostic offline data through a skill extractor (**yellow**) and low-level skill policy (**blue**). Also trains a prior distribution over skill embeddings (**green**). **(2) Skill-based Meta-training:** Meta-trains a high-level skill policy (**red**) and task encoder (**purple**) while using the pre-trained low-level policy. The pre-trained skill prior is used to regularize the high-level policy during meta-training and guide exploration. **(3) Target Task Learning:** Leverages the meta-trained hierarchical policy for quick learning of an unseen target task. After conditioning the policy by encoding a few transitions c^* from the target task \mathcal{T}^* , we continue fine-tuning the high-level skill policy on the target task while regularizing it with the pre-trained skill prior.

new tasks. Specifically, PEARL leverages the meta-training tasks for learning a task encoder $q(e|c)$. This encoder takes in a small set of state-action-reward transitions c and produces a task embedding e . This embedding is used to condition the actor $\pi(a|s, z)$ and critic $Q(s, a, e)$. In PEARL, actor, critic and task encoder are trained by jointly maximizing the obtained reward and the policy's entropy \mathcal{H} [104]:

$$\max_{\pi} \mathbb{E}_{\mathcal{T} \sim p_{\mathcal{T}}, e \sim q(\cdot | c^{\mathcal{T}})} \left[\sum_t \mathbb{E}_{(s_t, a_t) \sim \rho_{\pi|e}} [r_{\mathcal{T}}(s_t, a_t) + \alpha \mathcal{H}(\pi(a|s_t, e))] \right]. \quad (5.2)$$

Additionally, the task embedding output of the task encoder is regularized towards a constant prior distribution $p(e)$.

5.4 Approach

We propose **Skill-based Meta-Policy Learning (SiMPL)**, an algorithm for jointly leveraging offline data as well as a set of meta-training tasks to accelerate the learning of unseen target tasks. Our method has three phases: **(1) skill extraction:** we extract reusable skills and a skill prior from the offline data (Section 5.4.1),

(2) **skill-based meta-training**: we utilize the meta-training tasks to learn how to leverage the extracted skills and skill prior to efficiently solve new tasks (Section 5.4.2), (3) **target task learning**: we fine-tune the meta-trained policy to rapidly adapt to solve an unseen target task (Section 5.4.3). An illustration of the proposed method is shown in Figure 5.2.

5.4.1 Skill Extraction

To acquire a set of reusable skills from the offline dataset \mathbf{D} , we leverage the skill extraction approach proposed in Pertsch, Lee, and Lim [232]. Concretely, we jointly train (1) a skill encoder $q(z|s_{0:K}, a_{0:K-1})$ that embeds an K -steps trajectory randomly cropped from the sequences in \mathbf{D} into a low-dimensional skill embedding z , and (2) a low-level skill policy $\pi(a_t|s_t, z)$ that is trained with behavioral cloning to reproduce the action sequence $a_{0:K-1}$ given the skill embedding. To learn a smooth skill representation, we regularize the output of the skill encoder with a unit Gaussian prior distribution, and weight this regularization by a coefficient β [113]:

$$\max_{q, \pi} \mathbb{E}_{z \sim q} \left[\underbrace{\prod_{t=0}^{K-1} \log \pi(a_t|s_t, z)}_{\text{behavioral cloning}} - \underbrace{\beta D_{\text{KL}}(q(z|s_{0:K}, a_{0:K-1}), \mathcal{N}(0, I))}_{\text{embedding regularization}} \right]. \quad (5.3)$$

Additionally, we follow Pertsch, Lee, and Lim [232] and learn a skill prior $p(z|s)$ that captures the distribution of skills likely to be executed in a given state under the training data distribution. The prior is trained to match the output of the skill encoder: $\min_p D_{\text{KL}}([q(z|s_{0:K}, a_{0:K-1})], p(z|s_0))$. Here $[\cdot]$ indicates that gradient flow is stopped into the skill encoder for training the skill prior.

5.4.2 Skill-based Meta-Training

We aim to learn a policy that can quickly learn to leverage the extracted skills to solve new tasks. We leverage off-policy meta-RL (see Section 5.3) to learn such a policy using our set of meta-training tasks \mathbf{T} .

Similar to PEARL [243], we train a task-encoder that takes in a set of sampled transitions and produces a task embedding e . Crucially, we leverage our learned skills by training a task-embedding-conditioned policy over *skills* instead of primitive actions: $\pi(z|s, e)$, thus equipping the policy with a set of useful pre-trained behaviors and reducing the meta-training task to learning how to combine these behaviors instead of learning them from scratch. We find that this usage of offline data through learned skills is crucial for enabling meta-training on complex, long-horizon tasks (see Section 5.5).

Prior work has shown that the efficiency of RL on learned skill spaces can be substantially improved by guiding the policy with a learned skill prior [232, 4]. Thus, instead of regularizing with a maximum entropy objective as done in prior work on off-policy meta-RL [243], we propose to regularize the meta-training policy with our pre-trained skill prior, leading to the following meta-training objective:

$$\max_{\pi} \mathbb{E}_{\mathcal{T} \sim p_{\mathcal{T}}, e \sim q(\cdot | c^{\mathcal{T}})} \left[\sum_t \mathbb{E}_{(s_t, z_t) \sim \rho_{\pi|e}} [r_{\mathcal{T}}(s_t, z_t) - \alpha D_{\text{KL}}(\pi(z|s_t, e), p(z|s_t))] \right]. \quad (5.4)$$

where α determines the strength of the prior regularization. We automatically tune α via dual gradient descent by choosing a target divergence δ between policy and prior [232].

To compute the task embedding e , we used multiple different sizes of c . We found that we can increase training stability by adjusting the strength of the prior regularization to the size of the conditioning set. Intuitively, when the high-level policy is conditioned on only a few transitions, i.e. when the set c is small, it has only little information about the task at hand and should thus be regularized stronger towards the task-agnostic skill prior. Conversely, when c is large, the policy likely has more information about the target task and thus should be allowed to deviate from the skill prior more to solve the task, i.e. have a weaker regularization strength.

To implement this intuition, we employ a simple approach: we define *two* target divergences δ_1 and δ_2 and associated auto-tuned coefficients α_1 and α_2 with $\delta_1 < \delta_2$. We regularize the policy using the

larger coefficient α_1 with small conditioning transition set and otherwise we regularize using the smaller coefficient α_2 . We found this technique simple yet sufficient in our experiments and leave the investigation of more sophisticated regularization approaches for future work.

5.4.3 Target Task Learning

When a target task is given, we aim to leverage the meta-trained policy for quickly learning how to solve it. Intuitively, the policy should first explore different skill options to learn about the task at hand and then rapidly narrow its output distribution to those skills that solve the task. We implement this intuition by first collecting a small set of conditioning transitions c^* from the target task by exploring with the meta-trained policy. Since we have no information about the target task at this stage, we explore the environment by conditioning our pre-trained policy with task embeddings sampled from the task prior $p(e)$. Then, we encode this set of transitions into a target task embedding $e^* \sim q(e|c^*)$. By conditioning our meta-trained high-level policy on this encoding, we can rapidly narrow its skill distribution to skills that solve the given target task: $\pi(z|s, e^*)$.

Empirically, we find that this policy is often already able to achieve high success rates on the target task. Note that only very few interactions with the environment for collecting c^* are required for learning a complex, long-horizon and unseen target task with sparse reward. This is substantially more efficient than prior approaches such as SPiRL that require orders of magnitude more target task interactions for achieving comparable performance.

To further improve the performance on the target task, we fine-tune the conditioned policy with target task rewards while guiding its exploration with the pre-trained skill prior^{*}:

^{*}Other regularization distributions are possible during fine-tuning, e.g. the high-level policy conditioned on task prior samples $p(z|s, e \sim p(e))$ or the target task embedding conditioned policy $p(z|s, e^*)$ before finetuning. Yet, we found the regularization with the pre-trained task-agnostic skill prior to work best in our experiments.

$$\max_{\pi} \mathbb{E}_{e^* \sim q(\cdot | c^*)} \left[\sum_t \mathbb{E}_{(s_t, z_t) \sim \rho_{\pi|e^*}} [r_{\mathcal{T}^*}(s_t, z_t) - \alpha D_{\text{KL}}(\pi(z|s_t, e^*), p(z|s_t))] \right]. \quad (5.5)$$

In practice, we propose several techniques for stabilizing meta-training and fine-tuning: (1) adaptively regularizing the policy based on the size of the conditioning trajectory set as described in Section 5.4.2, (2) parameterizing the policy as a residual policy that outputs differences to the pre-trained skill prior instead of the approach from Pertsch, Lee, and Lim [232] that directly fine-tunes the skill prior, and (3) initializing the Q-function and α parameter during fine-tuning with meta-trained parameters instead of randomly initialized networks. We discuss these techniques in detail in Section 5.7.4.

5.5 Experiments

Our experiments aim to answer the following questions: (1) Can our proposed method learn to efficiently solve long-horizon, sparse reward tasks? (2) Is it crucial to utilize offline datasets to achieve this? (3) How can we best leverage the training tasks for efficient learning of target tasks? (4) How does the training task distribution affect the target task learning?

5.5.1 Experimental Setup

We evaluate our approach in two challenging continuous control environments: maze navigation and kitchen manipulation environment, as illustrated in Figure 5.3. While meta-RL algorithms are typically evaluated on tasks that span only a few dozen time steps and provide dense rewards [77, 252, 243, 354], our tasks require to learn long-horizon behaviors over hundreds of time steps from sparse reward feedback and thus pose a new challenge to meta-learning algorithms.

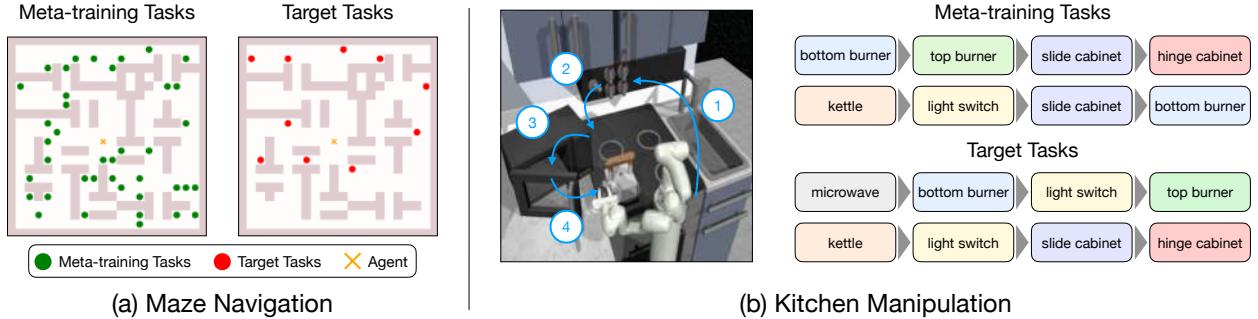


Figure 5.3: Environments. We evaluate our proposed framework in two domains that require the learning of complex, long-horizon behaviors from sparse rewards. These environments are substantially more complex than those typically used to evaluate meta-RL algorithms. (a) **Maze Navigation**: The agent needs to navigate for hundreds of steps to reach unseen target goals and only receives a binary reward upon task success. (b) **Kitchen Manipulation**: The 7DoF agent needs to execute an unseen sequence of four subtasks, spanning hundreds of time steps, and only receives a sparse reward upon completion of each subtask.

5.5.1.1 Maze Navigation

Environment. This 2D maze navigation domain based on the maze navigation problem in Fu et al. [87] requires long-horizon control with hundreds of steps for a successful episode and only provides sparse reward feedback upon reaching the goal. The observation space of the agent consists of its 2D position and velocity and it acts via planar, continuous velocity commands.

Offline Dataset & Meta-training / Target Tasks. Following Fu et al. [87] we collect a task-agnostic offline dataset by randomly sampling start-goal locations in the maze and using a planner to generate a trajectory that reaches from start to goal. Note that the trajectories are not annotated with any reward or task labels (*i.e.* which start-goal location is used for producing each trajectory). To generate a set of meta-training and target tasks, we fix the agent’s initial position in the center of the maze and sample 40 random goal locations for meta-training and another set of 10 goals for target tasks. All meta-training and target tasks use the same sparse reward formulation. More details can be found in Section 5.7.6.1.

5.5.1.2 Kitchen Manipulation

Environment. The FrankaKitchen environment of Gupta et al. [101] requires the agent to control a 7-DoF robot arm via continuous joint velocity commands and complete a sequence of manipulation tasks like opening the microwave or turning on the stove. Successful episodes span 300-500 steps and the agent is only provided a sparse reward signal upon successful completion of a subtask.

Offline Dataset & Meta-training / Target Tasks. We leverage a dataset of 600 human-teleoperated manipulation sequences of Gupta et al. [101] for offline pre-training. In each trajectory, the robot executes a sequence of four subtasks. We then define a set of 23 meta-training tasks and 10 target tasks that in turn require the consecutive execution of four subtasks (see Figure 5.3 for examples). Note that each task consists of a unique combination of subtasks. More details can be found in Section 5.7.6.2.

5.5.2 Baselines

We compare SiMPL to prior approaches in RL, skill-based RL, meta-RL, and multi-task RL.

- **SAC** [104] is a state of the art deep RL algorithm. It learns to solve a target task from scratch without leveraging the offline dataset nor the meta-training tasks.
- **SPiRL** [232] is a method designed to leverage offline data through the transfer of learned skills. It acquires skills and a skill prior from the offline dataset but does not utilize the meta-training tasks. This investigates the benefits our method can obtain from leveraging the meta-training tasks.
- **PEARL** [243] is a state of the art off-policy meta-RL algorithm that learns a policy which can quickly adapt to unseen test tasks. It learns from the meta-training tasks but does not use the offline dataset. This examines the benefits of using learned skills in meta-RL.
- **PEARL-ft** demonstrates the performance of a PEARL [243] model further fine-tuned on a target task using SAC [104].

- **Multi-task RL (MTRL)** is a multi-task RL baseline which learns from the meta-training tasks by distilling individual policies specialized in each task into a shared policy, similar to Distral [294]. Each individual policy is trained using SPiRL by leveraging skills extracted from the offline dataset. Therefore, it utilizes both the meta-training tasks and offline dataset similar to our method. This provides a direct comparison of multi-task learning (MTRL) from the training tasks vs. meta-learning using them (ours).

More implementation details on the baselines can be found in Section 5.7.5.

5.5.3 Results

We present the quantitative results in Figure 5.4 and the qualitative results on the maze navigation domain in Figure 5.5. In Figure 5.4, SiMPL demonstrates much better sample efficiency for learning the unseen target tasks compared to all the baselines. Without leveraging the offline dataset and meta-training tasks, SAC is not able to make learning progress on most of the target tasks. While PEARL is first trained on the meta-training tasks, it still achieves poor performance on the target tasks and fine-tuning it (PEARL-ft) does not yield significant improvement. We believe this is because both environments provide only sparse rewards yet require the model to exhibit long-horizon and complex behaviors, which is known to be difficult for meta-RL methods [196].

On the other hand, by first extracting skills and acquiring a skill prior from the offline dataset, SPiRL’s performance consistently improves with more samples from the target tasks. Yet, it requires significantly more environment interactions than our method to solve the target tasks since the policy is optimized using vanilla RL, which is not designed to learn to quickly learn new tasks. While the multi-task RL (MTRL) baseline first learns a multi-task policy from the meta-training tasks, its sample efficiency is similar to SPiRL on target task learning, which highlights the strength of our proposed method – meta-learning from the meta-training tasks for fast target task learning.

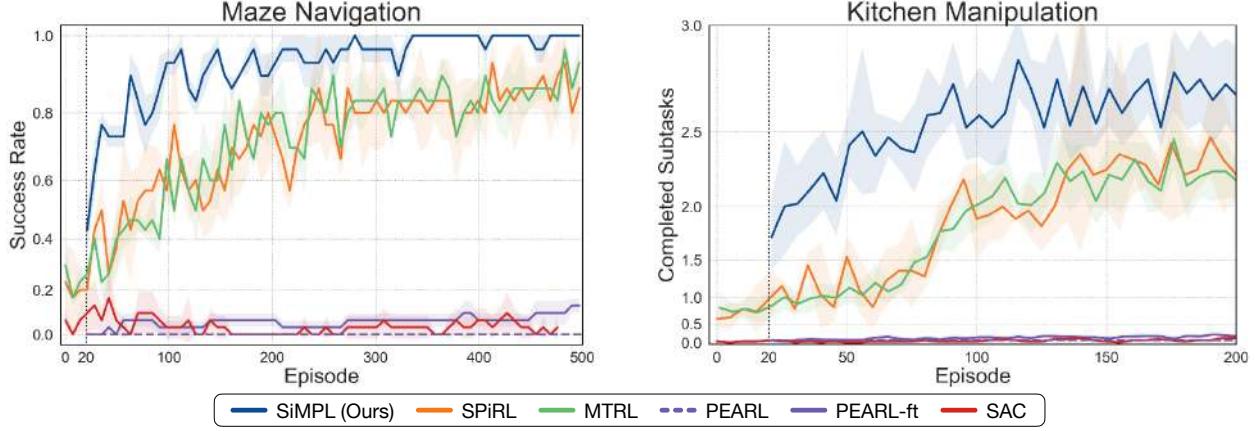


Figure 5.4: Target Task Learning Efficiency. SiMPL demonstrates better sample efficiency compared to all the baselines, verifying the efficacy of meta-learning on long-horizon tasks by leveraging skills and skill prior extracted from an offline dataset. For both the two environments, we train each model on each target task with 3 different random seeds. SiMPL and PEARL-ft first collect 20 episodes of environment interactions (vertical dotted line) for conditioning the meta-trained policy before fine-tuning it on target tasks.

Compared to the baselines, our method learns the target tasks much quicker. Within only a few episodes the policy converges to solve more than 80% of the target tasks in the maze environment and two out of four subtasks in the kitchen manipulation environment. The prior-regularized fine-tuning then continues to improve performance. The rapidly increasing performance and the overall faster convergence show the benefits of leveraging meta-training tasks in addition to learning from offline data: by first learning to learn how to quickly solve tasks using the extracted skills and the skill prior, our policy can efficiently solve the target tasks.

The qualitative results presented in Figure 5.5 show that all the methods that leverage the offline dataset (*i.e.* SiMPL, SPiRL, and MTRL) effectively explore the maze in the first episode. Then, SiMPL converges with much fewer episodes compared to SPiRL and MTRL, underlining the effectiveness of meta-training. In contrast, PEARL-ft is not able to make learning progress, justifying the necessity of employing offline datasets for acquiring long-horizon, complex behaviors.

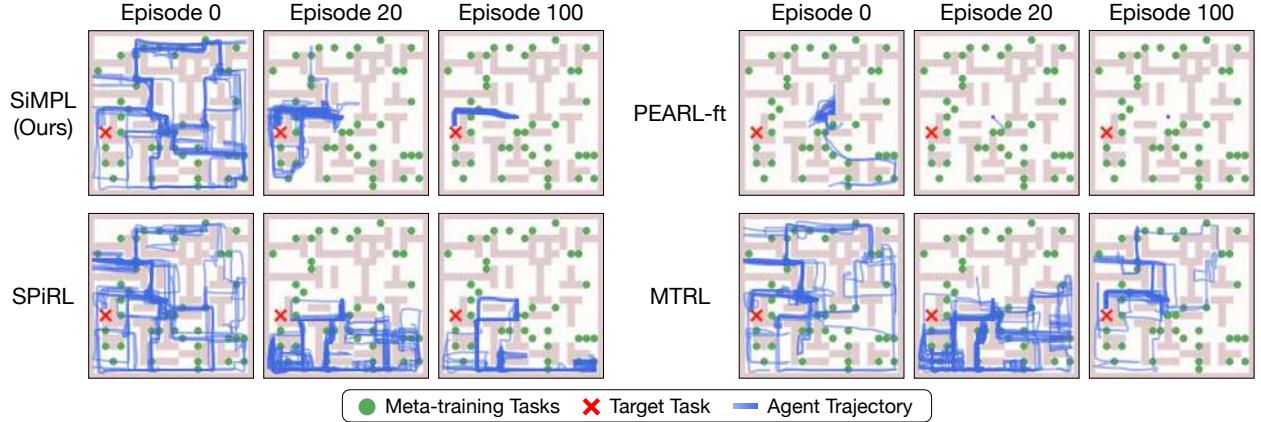


Figure 5.5: **Qualitative Results.** All the methods that leverage the offline dataset (*i.e.* SiMPL, SPiRL, and MTRL) effectively explore the maze in the first episode. Then, SiMPL converges with much fewer episodes compared to SPiRL and MTRL. In contrast, PEARL-ft is not able to make learning progress.

5.5.4 Meta-Training Task Distribution Analysis

In this section, we aim to investigate the effect of the meta-training task distribution on our skill-based meta-training and target task learning phases. Specifically, we examine the effect of (1) the number of tasks in the meta-training task distribution and (2) the alignment between a meta-training task distribution and target task distribution. We conduct experiments and analyses in the maze navigation domain. More details on task distributions can be found in Section 5.7.6.1.

Number of meta-training tasks. To investigate how the number of meta-training tasks affects the performance of our method, we train our method with fewer numbers meta-training tasks (*i.e.* 10 and 20) and evaluate it with the same set of target tasks. The quantitative results presented in Figure 5.6(a) suggest that even with sparser meta-training task distributions (*i.e.* fewer numbers of meta-training tasks), SiMPL is still more sample efficient compared to the best-performing baseline (*i.e.* SPiRL).

Meta-train / test task alignment. We aim to examine if a model trained on a meta-training task distribution that aligns better/worse with the target tasks would yield improved/deteriorated performance. To this end, we create biased meta-training / test task distributions: we create a meta-train set by sampling goal locations from only the top 25% portion of the maze ($\mathcal{T}_{\text{TRAIN-TOP}}$). To rule out the effect of the density of the task

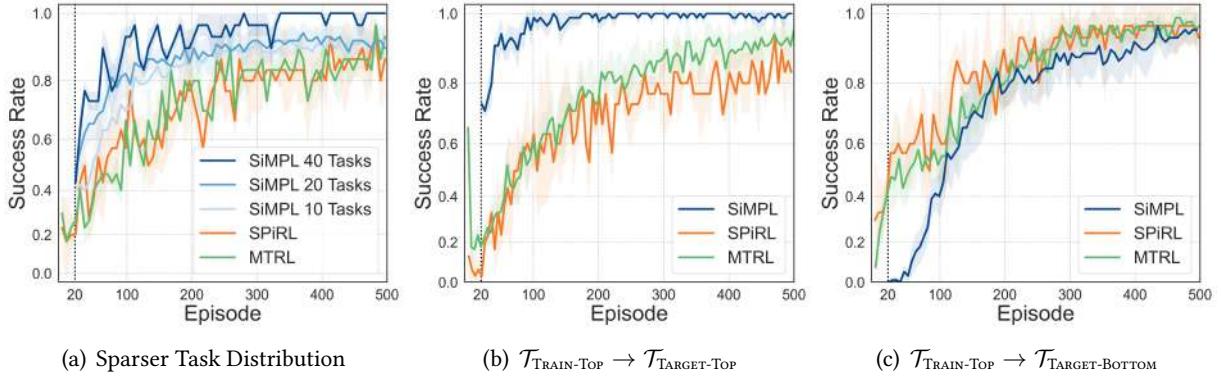


Figure 5.6: Meta-training Task Distribution Analysis. (a) With sparser meta-training task distributions (*i.e.* fewer numbers of meta-training tasks), SiMPL still achieves better sample efficiency compared to SPIRL, highlighting the benefit of leveraging meta-training tasks. (b) When trained on a meta-training task distribution that aligns better with the target task distribution, SiMPL achieves improved performance. (c) When trained on a meta-training task distribution that is mis-aligned with the target tasks, SiMPL yields worse performance. For all the analyses, we train each model on each target task with 3 different random seeds.

distribution, we sample 10 (*i.e.* $40 \times 25\%$) meta-training tasks. Then, we create two target task distributions that have good and bad alignment with this meta-training distribution respectively by sampling 10 target tasks from the top 25% portion of the maze ($\mathcal{T}_{\text{TARGET-TOP}}$) and 10 target tasks from the bottom 25% portion of the maze ($\mathcal{T}_{\text{TARGET-BOTTOM}}$).

Figure 5.6(b) and Figure 5.6(c) present the target task learning efficiency for models trained with good task alignment (meta-train on $\mathcal{T}_{\text{TRAIN-TOP}}$, learn target tasks from $\mathcal{T}_{\text{TARGET-TOP}}$) and bad task alignment (meta-train on $\mathcal{T}_{\text{TRAIN-TOP}}$, learn target tasks from $\mathcal{T}_{\text{TARGET-BOTTOM}}$), respectively. The results demonstrate that SiMPL can achieve improved performance when trained on a better aligned meta-training task distribution. On the other hand, not surprisingly, SiMPL and MTRL perform slightly worse compared to SPIRL when trained with misaligned meta-training tasks (see Figure 5.6(c)). This is expected given that SPIRL does not learn from the misaligned meta-training tasks. In summary, from Figure 5.6, we can conclude that meta-learning from either a diverse task distribution or a better informed task distribution can yield improved performance for our method.

5.6 Conclusion

We propose a skill-based meta-RL method, dubbed SiMPL, that can meta-learn on long-horizon tasks by leveraging prior experience in the form of large offline datasets without additional reward and task annotations. Specifically, our method first learns to extract reusable skills and a skill prior from the offline data. Then, we propose to meta-train a high-level policy that leverages these skills for efficient learning of unseen target tasks. To effectively utilize learned skills, the high-level policy is regularized by the acquired prior. The experimental results on challenging continuous control long-horizon navigation and manipulation tasks with sparse rewards demonstrate that our method outperforms the prior approaches in deep RL, skill-based RL, meta-RL, and multi-task RL. In the future, we aim to demonstrate the scalability of our method to high-DoF continuous control problems on real robotic systems to show the benefits of our improved sample efficiency.

5.7 Appendix

5.7.1 Meta-Reinforcement Learning Method Ablation

In this section, we compare the learning efficiency of different meta-RL algorithms with respect to the length of the training tasks. Specifically, we hypothesize that our approach SiMPL, which extracts temporally extended skills from offline experience, is better suited for learning long-horizon tasks than prior meta-RL algorithms. To cleanly investigate the importance of the temporally extended skills vs. the importance of using prior experience we include two additional comparisons to methods that leverage prior experience for meta-RL but via flat behavioral cloning instead of through temporally extended skills:

- **BC+PEARL** first learns a behavior cloning (BC) policy through supervised learning from the offline dataset. Then, analogous to our approach SiMPL, during the meta-training phase, a task encoder

and a meta-learned policy are meta-trained with the BC policy constrained SAC objective. For fair comparison, we use the same residual policy parameterization as described in Section 5.7.4.1.

- **BC+MAML** follows the same learning procedure described above, but uses MAML [77] for meta-training instead of PEARL. We follow the original learning objective in Finn, Abbeel, and Levine [77] (*i.e.* using REINFORCE [324] for task adaptation, and using TRPO [264] for meta-policy optimization).

We compare these methods as well as the standard meta-RL approach PEARL [243] on three meta-training tasks distributions of increasing complexity in the maze navigation environment (see Figure 5.7): (1) short-range goals with small variance $\mathcal{T}_{\text{TRAIN-EASY}}$, (2) short-range goals with larger variance $\mathcal{T}_{\text{TRAIN-MEDIUM}}$, and (3) long-range goals with large variance $\mathcal{T}_{\text{TRAIN-HARD}}$, which we used in our original maze experiments. By increasing variance and length of the tasks in each task distribution, we can investigate the learning capability of the meta-RL algorithms.

We present the quantitative results in Figure 5.8 and the corresponding qualitative analysis in Figure 5.9. On the simplest task distribution we find that all approaches can learn to solve the tasks efficiently, except for BC+MAML. While the latter also learns to solve the task eventually (see performance upon convergence as dashed orange line in Figure 5.8(a)) it uses on-policy meta-RL and thus requires substantially more environment interactions during meta-training. We thus only consider the more sample efficient BC+PEARL off-policy meta-RL method in the remaining comparisons.

On the more complex task distributions $\mathcal{T}_{\text{TRAIN-MEDIUM}}$ and $\mathcal{T}_{\text{TRAIN-HARD}}$, we find that using prior data for meta-learning is generally beneficial: both BC+PEARL and SiMPL learn more efficiently on the task distribution of medium difficulty $\mathcal{T}_{\text{TRAIN-MEDIUM}}$, as shown in Figure 5.8(b), since the policy pre-trained from offline data allows for more efficient exploration during meta-training. Importantly, on the hardest task distribution $\mathcal{T}_{\text{TRAIN-HARD}}$, as shown in Figure 5.8(c), which consists exclusively of long-horizon tasks, we find that only SiMPL is able to effectively learn, highlighting the importance of leveraging the offline data via

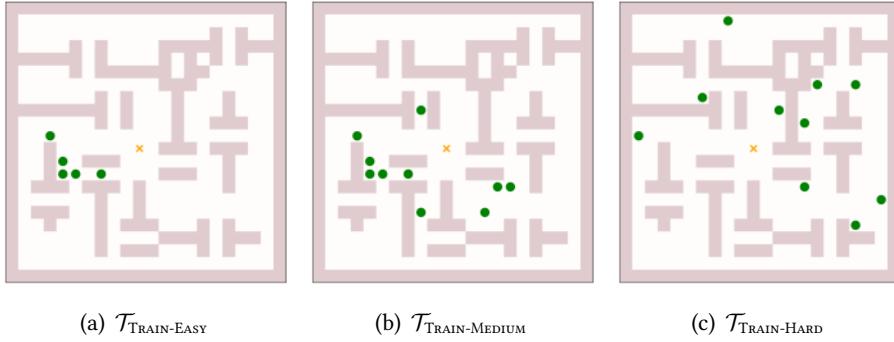


Figure 5.7: Task Distributions for Task Length Ablation. We propose three meta-training task distributions of increasing difficulty to compare different meta-RL algorithms: $\mathcal{T}_{\text{TRAIN-EASY}}$ uses short-horizon tasks with adjacent goal locations, making exploration easier during meta-training, $\mathcal{T}_{\text{TRAIN-MEDIUM}}$ uses similar task horizon but increases the goal position variance, $\mathcal{T}_{\text{TRAIN-HARD}}$ contains long-horizon tasks with high variance in goal position and thus is the hardest of the tested task distributions.

temporally extended skills instead of flat behavioral cloning. This supports our intuition that the abstraction provided by skills is particularly beneficial for meta-learning on long-horizon tasks.

5.7.2 Learning Efficiency on Target Tasks with Few Episodes of Experience

In this section, we examine the data efficiency of the compared methods on the target tasks, specifically when provided with only a *few* (<20) episodes of online interaction with an unseen target task. Being able to learn new tasks this quickly is a major strength of meta-RL approaches [77, 243]. We hypothesize that our skill-based meta-RL algorithm SiMPL can learn similarly fast, even on long-horizon, sparse-reward tasks.

In our original evaluations in Section 5.5, we used 20 episodes of initial exploration to condition our meta-trained policy. In Figure 5.10, we instead compare performance of different approaches when only provided with very few episodes of online interactions. We find that SiMPL learns to solve the unseen tasks substantially faster than all alternative approaches. On the kitchen manipulation tasks our approach learns to almost solve two out of four subtasks within a time span equivalent to only a few minutes of real-robot execution time. In contrast, prior meta-RL methods struggle at making progress at all on such long-horizon tasks, showing the benefit of combining meta-RL with prior offline experience.

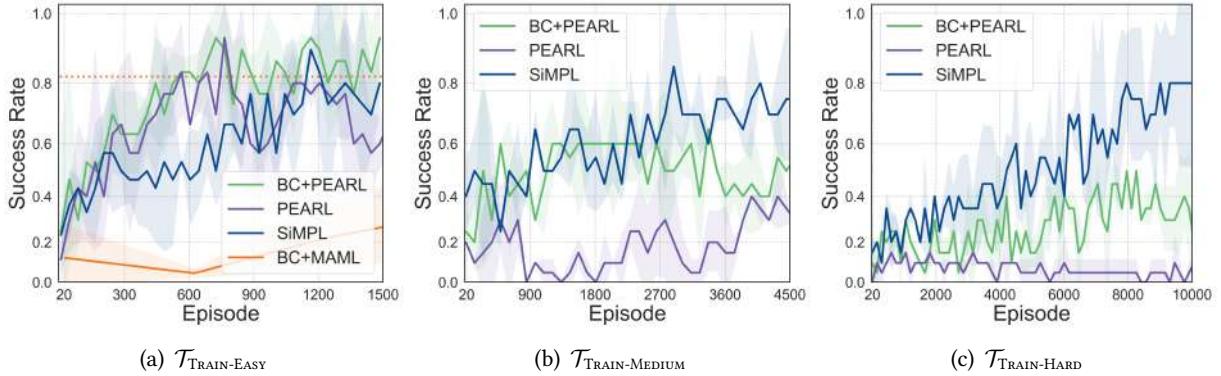


Figure 5.8: Meta-Training Performance for Task Length Ablation. We find that most meta-learning approaches can solve the simplest task distribution, but using prior experience in BC+PEARL and SiMPL helps for the more challenging distributions (b) and (c). We find that only our approach, which uses the prior data by extracting temporally extended skills, is able to learn the challenging long-horizon tasks efficiently.

5.7.3 Investigating Offline Data vs. Target Domain Shift

To provide more insights on comparing SiMPL and SPiRL [232], we evaluate SiMPL in the maze navigation task setup proposed in Pertsch, Lee, and Lim [232]. This tests whether our approach can scale to image-based observations: Pertsch, Lee, and Lim [232] use 32×32 px observations centered around the agent. Even more importantly, it allows us to investigate the robustness of the approach to the domain shifts between the offline pre-training data and the target task: we use the maze navigation offline dataset from Pertsch, Lee, and Lim [232] which was collected on *randomly sampled* 20×20 maze layouts and test on tasks in the unseen, randomly sampled 40×40 test maze layout from Pertsch, Lee, and Lim [232]. We visualize the meta-training task distribution in Figure 5.11(a) and the target task distribution in Figure 5.11(b).

We compare the performance of our method to the best-performing baseline, SPiRL [232], in Figure 5.11(c). Similar to the result presented in Figure 5.4, SiMPL can learn the target task faster by combining skills learned from the offline dataset with efficient meta-training. This shows that our approach can scale to image-based inputs and is robust to substantial domain shifts between the offline pre-training data and the target tasks.

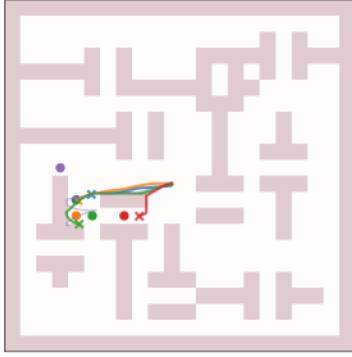
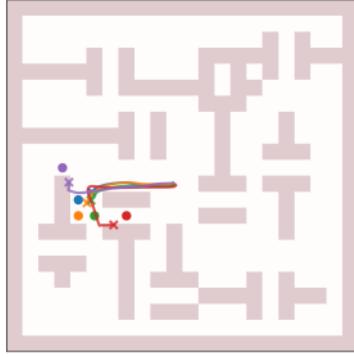
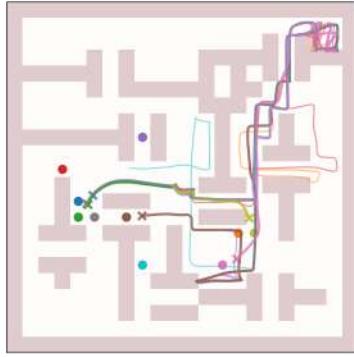
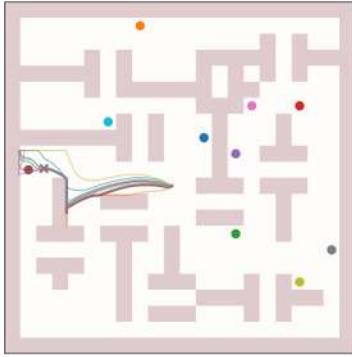
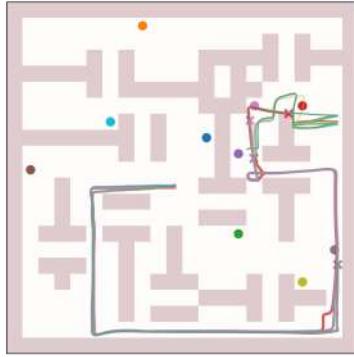
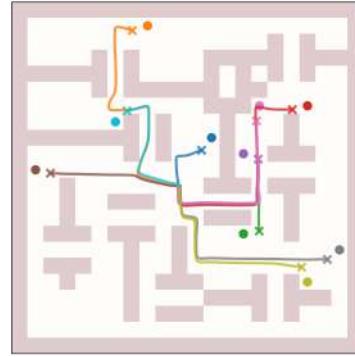
(a) PEARL on $\mathcal{T}_{\text{TRAIN-EASY}}$ (b) BC+PEARL on $\mathcal{T}_{\text{TRAIN-EASY}}$ (c) SiMPL on $\mathcal{T}_{\text{TRAIN-EASY}}$ (d) PEARL on $\mathcal{T}_{\text{TRAIN-MEDIUM}}$ (e) BC+PEARL on $\mathcal{T}_{\text{TRAIN-MEDIUM}}$ (f) SiMPL on $\mathcal{T}_{\text{TRAIN-MEDIUM}}$ (g) PEARL on $\mathcal{T}_{\text{TRAIN-HARD}}$ (h) BC+PEARL on $\mathcal{T}_{\text{TRAIN-HARD}}$ (i) SiMPL on $\mathcal{T}_{\text{TRAIN-HARD}}$

Figure 5.9: Qualitative Result of Meta-RL Method Ablation. **Top.** All the methods can learn to solve short-horizon tasks $\mathcal{T}_{\text{TRAIN-EASY}}$. **Middle.** On medium-horizon tasks $\mathcal{T}_{\text{TRAIN-MEDIUM}}$, PEARL struggles at exploring further, while BC+PEARL exhibits more consistent exploration yet still fails to solve some of the tasks. SiMPL can explore well and solve all the tasks. **Bottom.** On long-horizon tasks $\mathcal{T}_{\text{TRAIN-HARD}}$, PEARL falls into a local minimum, focusing only on one single task on the left. BC+PEARL explores slightly better and can solve a few more tasks. SiMPL can effectively learn all the tasks.

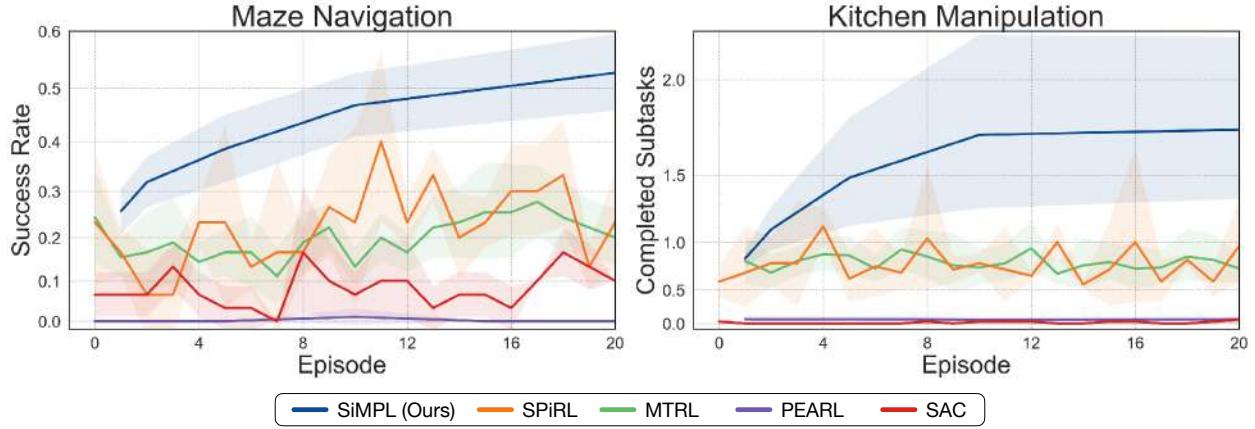


Figure 5.10: Performance with Few Episodes of Target Task Interaction. We find that our skill-based meta-RL approach SiMPL is able to learn complex, long-horizon tasks within few episodes of online interaction with a new task while prior meta-RL approaches and non-meta-learning baselines require many more interactions or fail to learn the task altogether.

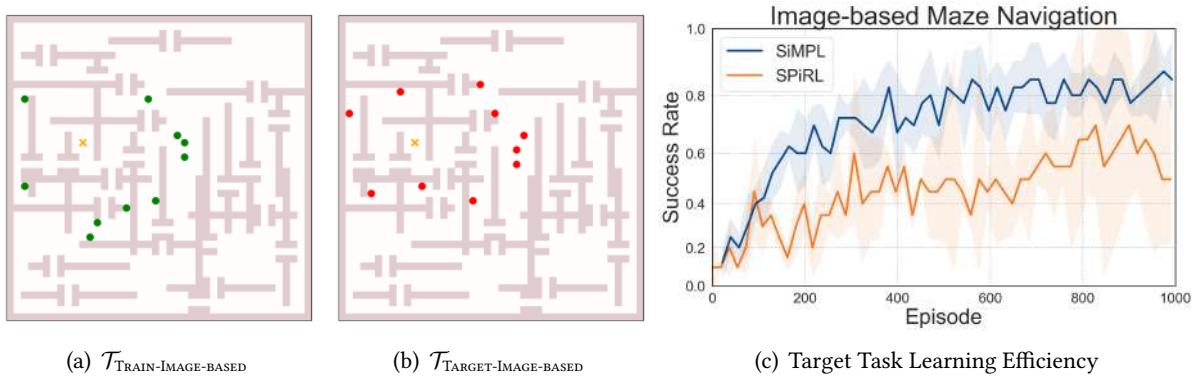


Figure 5.11: Image-Based Maze Navigation with Distribution Shift. (a-b): Meta-training and target task distributions. The green dots represent the goal locations of meta-training tasks and the red dots represent the goal locations of target tasks. The yellow cross represent the initial location of the agent, which is equivalent to the one used in Pertsch, Lee, and Lim [232]. (c): Performance on the target task. Our approach SiMPL can leverage skills learned from offline data for efficient meta-RL on the maze navigation task and is robust to the domain shift between offline data environments and the target environment.

5.7.4 Implementation Details on Our Method

In this section, we describe the additional implementation details on our proposed method. The details on model architecture is presented in Section 5.7.4.1, followed by the training detailed described in Section 5.7.4.2.

5.7.4.1 Model Architecture

We describe the details on our model architecture in this section.

Skill Prior We followed architecture and learning procedure of Pertsch, Lee, and Lim [232] for learning a low-level skill policy and a skill prior. Please refer to Pertsch, Lee, and Lim [232] for more details on the architectures for learning skills and skill priors from offline datasets.

Task Encoder Following Rakelly et al. [243], our task encoder is a permutation invariant neural network. Specifically, we adopt Set Transformer [157] that consists of layers $[2 \times \text{ISAB}_{32} \rightarrow \text{PMA}_1 \rightarrow 3 \times \text{MLP}]$ for expressive and efficient set encoding. All the hidden layers are 128-dimensional and all attention layers have 4 attention heads. The encoder takes a set of high-level transitions as input, where each transition is a vector concatenation of high-level transition tuple. The output of the encoder is (μ_e, σ_e) which are the parameters of Gaussian task posterior $p(e|c) = \mathcal{N}(e; \mu_e, \sigma_e)$. We varied task vector dimension $\text{dim}(e)$ depends on task distribution complexity. $\text{dim}(e) = 10$ for Kitchen Manipulation, $\text{dim}(e) = 6$ for Maze Navigation with 40 meta-training tasks, and $\text{dim}(e) = 5$, otherwise.

Policy We parameterize our policy with neural network. We employed 4-layer MLPs with 256 hidden units for Maze Navigation, and 6-layer MLPs with 128 hidden unit for Kitchen Manipulation experiment. Instead of direct parameterization of policy, the network output is added to skill-prior to make learning more stable. Specifically, the policy network takes concatenation of (s, e) as input, and then outputs residual

parameters $(\mu_z, \log \sigma_z)$ to skill-prior distribution $p(z|s) = \mathcal{N}(z|\mu_p, \sigma_p)$. Resulting distribution by this residual parameterization is $\pi(z|s) = \mathcal{N}(z|\mu_p + \mu_z, \exp(\log \sigma_p + \log \sigma_z))$

Critic The critic network takes concatenation of s, e , and skill z as input and outputs an estimation of task-conditioned Q-value $Q(s, z, e)$. We employ double Q networks [304] to mitigate Q-value overestimation. The architecture of critic follows the policy.

5.7.4.2 Training Details

For all the network updates, we used Adam optimizer [140] with a learning rate of $3e - 4$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$. We describe the training details of the skill-based meta-training phase in Section 5.7.4.2 and the target task learning phase Section 5.7.4.2.

Skill-based Meta-training Our meta-training procedure is similar to the procedure adopted in [243]. Encoder and critics networks are updated to minimize MSE between Q-value prediction and target Q value. Policy network is updated to optimize Equation 5.4 without updating the encoder network. All network are updated with the average gradients of 20 randomly sampled tasks. Each batch of gradients is computed from 1024 and 256 transitions for Maze Navigation and Kitchen Manipulation experiment, respectively. We train our models for 10000, 18000, and 16000 episodes for the Maze Navigation experiments with 10, 20, 40 meta-training tasks, respectively, and 3450 episodes for Kitchen Manipulation.

As stated in Section 5.4.2, we apply different regularization coefficients depending on the size of the conditioning transitions. In Maze Navigation experiment, we set target KL divergence to 0.1 for batch that is conditioned on size 4 transitions and 0.4 for batch conditioned on size 8192 transitions. In Kitchen Manipulation experiment, we set target KL divergence to 0.4 for bath conditioned with a size 1024 transitions while KL coefficient for batch conditioned on size 2 transitions is fixed to 0.3.

Target Task Learning We initialize the Q function and the auto-tuning value α with the values that learned in the skill-based meta-training phase. The policy is initialized after observing and encoding 20 episodes obtained from the task unconditioned policy rollouts. For the target task learning phase, the target KL δ is 1 for Maze Navigation, and 2 for Kitchen Manipulation experiments. To compute a gradient step, 256 high-level transitions are sampled from a replay buffer with size 20000. Note that we used same setup for baselines that uses SPiRL fine-tuning (SPiRL and MTRL).

5.7.5 Implementation Details on Baselines

In this section, we describe the additional implementation details on producing the results of the baselines.

5.7.5.1 SAC

The SAC [104] baseline learns to solve a target task from scratch without leveraging the offline dataset nor the meta-training tasks.

We initialize α to 0.1 and set the target entropy to $\mathcal{H} = -\dim(\mathcal{A})$. To compute a gradient step, 4096 and 1024 environment transitions are sampled from a replay buffer for Maze Navigation and Kitchen Navigation experiments, respectively.

5.7.5.2 PEARL and PEARL-ft

PEARL [243] learns from the meta-training tasks but does not use the offline dataset. Therefore, we directly train PEARL models on the meta-training tasks without the phase of learning from offline datasets. We use gradients averaged from 20 randomly sampled tasks where each task gradient is computed by batch sampled from a per-task buffer. The target entropy is set to $\mathcal{H} = -\dim(\mathcal{A})$ and α is initialized to 0.1.

While the method proposed in Rakelly et al. [243] does not fine-tune on target/meta-testing tasks, we extend PEARL to be fine-tuned on target tasks for a fair comparison, called PEARL-ft. Since PEARL does not use learned skills or a skill prior, the target task learning of PEARL is simply running SAC with

task-encoded initialization. Similar to the target task learning of our method, we initialize the Q function and entropy coefficient α to the value learned during the meta-training phase. Also, we initialize the policy to the task conditioned policy after observing 20 episodes of experience from the task unconditioned policy rollouts. The hyperparameters used for fine-tuning are the same as SAC.

5.7.5.3 SPiRL

Similar to our method, we initialize the high-level policy to skill-prior while fixing low-level policy for target task learning for SPiRL. α is initialized to 0.01 and we use the same hyperparameters for the SPiRL models as our method.

5.7.5.4 Multi-task RL (MTRL)

Inspired by Distral [294], our multi-task RL baseline is designed to first learn a set of individual policies, where each of them is specialized in one task; then, a shared/multi-task policy is learned by distilling the individual policies. Since it is inefficient to learn an individual policy from scratch, we learn each individual policy using SPiRL with learned skills and a skill prior. Then, we distill the individual policies using the following objective :

$$\max_{\pi_0} \mathbb{E}_{\mathcal{T} \sim p_{\mathcal{T}}} \left[\sum_t \mathbb{E}_{(s_t, z_t) \sim \rho_{\pi_0}} [r_{\mathcal{T}}(s_t, z_t) - \alpha D_{\text{KL}}(\pi_0(z|s_t, e), p(z|s_t))] \right]. \quad (5.6)$$

We use the same setup for α as our method, where α is auto-tuned to satisfy a target KL, $\delta = 0.1$ for Maze Navigation and $\delta = 0.2$ for Kitchen Manipulation.

While the target task learning phase for MTRL is similar to ours, except that MTRL is not initialized with a meta-trained Q function and learned α .

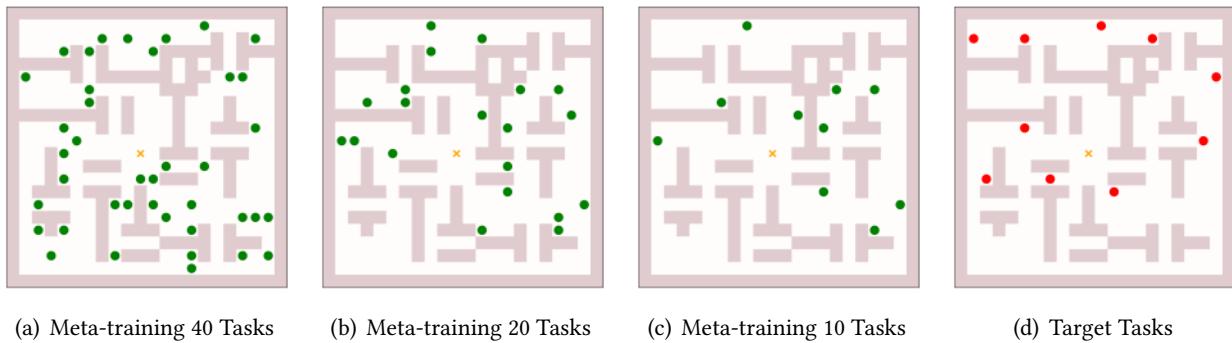


Figure 5.12: **Maze Meta-training and Target Task Distributions.** The green dots represent the goal locations of meta-training tasks and the red dots represent the goal locations of target tasks. The yellow cross represent the initial location of the agent.

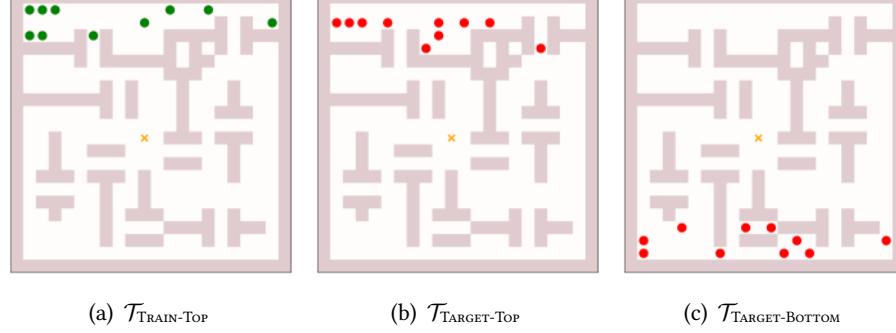


Figure 5.13: **Maze Meta-training and Target Task Distributions for Meta-training Task Distribution Analysis.** The green dots represent the goal locations of meta-training tasks and the red dots represent the goal locations of target tasks. The yellow cross represent the initial location of the agent.

5.7.6 Meta-Training Tasks and Target Tasks.

In this section, we present the meta-training tasks and target tasks used in the maze navigation domain and the kitchen manipulation domain.

5.7.6.1 Maze Navigation

The meta-training tasks and target tasks are visualized in Figure 5.12 and Figure 5.13.

5.7.6.2 Kitchen Manipulation

The meta-training tasks are:

- microwave → kettle → bottom burner → slide cabinet
- microwave → bottom burner → top burner → slide cabinet
- microwave → top burner → light switch → hinge cabinet
- kettle → bottom burner → light switch → hinge cabinet
- microwave → bottom burner → hinge cabinet → top burner
- kettle → top burner → light switch → slide cabinet
- microwave → kettle → slide cabinet → bottom burner
- kettle → light switch → slide cabinet → bottom burner
- microwave → kettle → bottom burner → top burner
- microwave → kettle → slide cabinet → hinge cabinet
- microwave → bottom burner → slide cabinet → top burner
- kettle → bottom burner → light switch → top burner
- microwave → kettle → top burner → light switch
- microwave → kettle → light switch → hinge cabinet
- microwave → bottom burner → light switch → slide cabinet
- kettle → bottom burner → top burner → light switch
- microwave → light switch → slide cabinet → hinge cabinet
- microwave → bottom burner → top burner → hinge cabinet
- kettle → bottom burner → slide cabinet → hinge cabinet

- bottom burner → top burner → slide cabinet → light switch
- microwave → kettle → light switch → slide cabinet
- kettle → bottom burner → top burner → hinge cabinet
- bottom burner → top burner → light switch → slide cabinet

The target tasks are:

- microwave → bottom burner → light switch → top burner
- microwave → bottom burner → top burner → light switch
- kettle → bottom burner → light switch → slide cabinet
- microwave → kettle → top burner → hinge cabinet
- kettle → bottom burner → slide cabinet → top burner
- kettle → light switch → slide cabinet → hinge cabinet
- kettle → bottom burner → top burner → slide cabinet
- microwave → bottom burner → slide cabinet → hinge cabinet
- bottom burner → top burner → slide cabinet → hinge cabinet
- microwave → kettle → bottom burner → hinge cabinet

Chapter 6

Learning from Observation

6.1 Introduction

Humans are effective at learning a task from demonstrations and applying the learned behaviors to other situations. We achieve this by extracting the underlying structure of the task when observing others fulfilling the task, instead of simply memorizing the demonstrator’s low-level actions [26, 116]. This high-level task structure generalizes to new situations and thus helps us to quickly learn the task in new situations. One intuitive and readily available instance of such high-level task structure is *task progress*, measuring how much of the task the agent completed. Inspired by this insight, we propose a novel imitation learning method that utilizes task progress for better generalization to unseen states and goals.

Typical learning from demonstration (LfD) approaches [236, 80] greedily imitate the expert policy and thus suffer from accumulated errors causing a drift away from states seen in the demonstrations [251]. To make the imitation policy more robust to states not in demonstrations, adversarial imitation learning methods [114, 88] encourage the agent to stay near the expert trajectories using a learned reward that distinguishes expert and agent behaviors. However, such learned reward functions often overfit to the expert demonstrations by learning spurious correlations between task-irrelevant features and expert/agent labels [355], and thus suffer from generalization to slightly different initial and goal configurations from the ones seen in the demonstrations (e.g. holdout goal regions or larger perturbation in goal sampling).

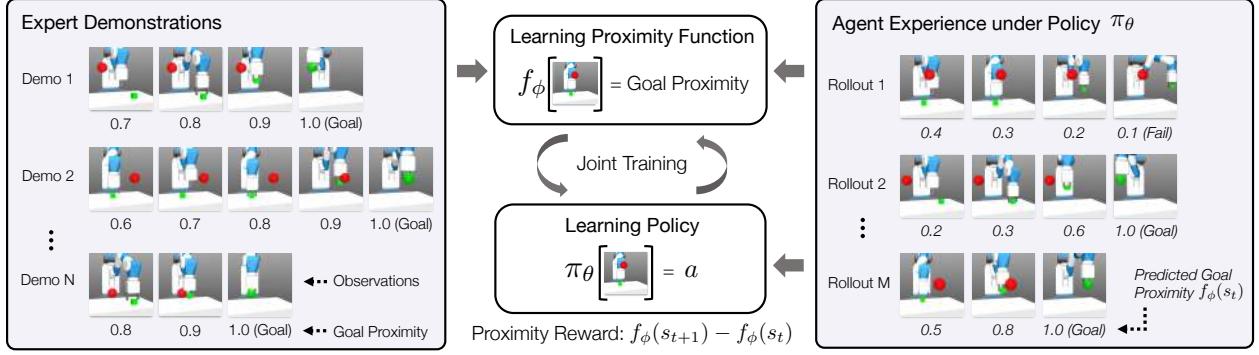


Figure 6.1: In goal-directed tasks, states on an expert trajectory have increasing proximity toward the goal as the expert makes progress towards fulfilling a task. Inspired by this intuition, we propose to learn a proximity function f_ϕ from expert demonstrations and agent experience, which predicts *goal proximity* (i.e. an estimate of temporal distance to the goal). Then, using this proximity function, we train a policy π_θ to progressively move to states with higher *predicted goal proximity* (*italicized numbers*) and eventually reach the goal. We alternate these two learning phases to improve both the proximity function and policy, leading to not only better generalization but also superior performance.

To learn a more generalizable and informative reward from demonstrations, we propose an imitation learning from observation (LfO) method, which learns a task progress estimator and uses the task progress estimate as a dense reward for training a policy as illustrated in Figure 6.1. Unlike discriminating expert and agent behaviors by predicting *binary* labels in prior adversarial imitation learning methods, which is prone to overfitting to task-irrelevant features, the task progress estimator is required to learn more task-relevant information to precisely predict the task progress on a *continuous* scale. Hence, it can generalize better to unseen states and provide more informative rewards.

As a measure of progress in goal-directed tasks, we define *goal proximity*, which is an estimate of temporal distance to the goal (i.e. the number of actions required to reach the goal) and entails all semantic information about how to reach the goal. We then train a *proximity function* to predict the goal proximity from expert demonstrations and agent experience. This proximity function acts as a dense reward to guide a reinforcement learning agent to reach states with high proximity, leading to the goal. In this paper, we focus on learning the proximity function and policy in a state space shared by the expert and learner, and leave generalizing to different embodiments as future work.

However, the predicted goal proximity can still be inaccurate on states not in the demonstrations, resulting in unstable policy learning. To improve the accuracy of the proximity function, we continually update it with trajectories from both the expert and learning agent. In addition, we penalize trajectories with the uncertainty of the proximity prediction to prevent the policy from exploiting inaccurate high proximity predictions. By leveraging the agent experience and predicting proximity function uncertainty, the proposed method achieves more efficient and stable policy learning.

The main contribution of this paper is an LfO algorithm for goal-directed tasks with better generalization to new goals or states not in demonstrations using goal proximity that informs an agent of the task progress. Together with a difference-based reward and uncertainty penalty of goal proximity estimation, our method provides more informative and robust rewards. Our extensive experiments show that the policy learned with the goal proximity function generalizes better than the state-of-the-art LfO algorithms on various goal-directed tasks, including navigation, locomotion, and robotic manipulation. Moreover, our method shows comparable results with LfD methods which learn from expert actions and a goal-conditioned imitation learning method which uses a sparse task reward.

6.2 Related Work

Imitation learning [258] aims to leverage expert demonstrations to acquire skills. While behavioral cloning [236] is simple but effective with a large number of demonstrations, it suffers from compounding errors caused by covariate shift [251]. On the other hand, inverse reinforcement learning (IRL) [210, 2, 352] estimates the underlying reward from demonstrations and trains a policy through reinforcement learning (RL) with this reward, which can better handle the compounding errors. Specifically, generative adversarial imitation learning (GAIL) [114] shows improved demonstration efficiency by training a discriminator to distinguish expert and agent transitions and using the discriminator output as a reward for policy training.

GoalGAIL [67] further improves sample efficiency for goal-directed tasks by relabeling transitions [14] and using true environment rewards.

While these imitation learning algorithms require expert actions, imitation learning from observation (LfO) approaches learn from state-only demonstrations, such as videos and kinesthetic demonstrations. To imitate demonstrations without expert actions, inverse dynamics models [213, 300, 225], reachability functions [159], or learned reward functions [74, 268, 267, 176] can be learned and used for policy training, but training such models requires a large amount of quality data or additional test-time demonstrations. On the other hand, state-only adversarial imitation learning [301] can imitate from a few demonstrations.

However, in such adversarial imitation learning approaches, the discriminator tends to find spurious associations between task-irrelevant features and expert/agent labels [355]. This becomes problematic when the agent encounters unseen states and the discriminator erroneously assigns agent behaviors low scores based on these task-irrelevant features, providing a poor reward for the agent. To overcome finding spurious associations, in addition to discriminating expert and agent trajectories, we propose to also estimate the proximity to the goal, which requires more task-relevant information and thus generalizes better to new states.

Temporal progress estimation has shown its effectiveness as an auxiliary reward for RL [188, 73, 160] and decision making criteria [51, 16, 36]. However, these methods learn the progress estimator only from the given demonstrations. This hinders policy learning when the progress estimator fails to generalize to agent experience, allowing the agent to exploit inaccurate progress predictions for higher reward. Moreover, greedily choosing an action with the highest predicted temporal progress [51, 16, 36] could lead to low long-term returns. By incorporating online updates, uncertainty estimates, and a difference-based proximity reward, our method robustly learns from demonstrations to solve goal-directed tasks without access to expert actions or the true environment reward.

6.3 Method

In this paper, we address the problem of LfO for goal-directed tasks with a focus on generalization to states or goals not covered in the demonstrations. Adversarial LfO methods [301, 335] suggest learning a reward function that penalizes agent state transitions deviating from the expert trajectories. However, these learned reward functions often focus on task-irrelevant features [355] and do not generalize to states not in the demonstrations, leading to unsuccessful policy training.

To learn a generalizable reward, we propose to leverage task progress information freely available in demonstrations, in terms of *goal proximity*, which estimates temporal distance to the goal (i.e. number of actions required to reach the goal). Predicting precise goal proximity on a continuous scale, rather than simply distinguishing expert and agent states, requires the model to capture task-relevant information, allowing the proximity prediction to generalize to states not in the demonstrations (Section 6.3.2). Then, a policy learns to reach states with higher proximity prediction, leading to the goal (Section 6.3.3). Moreover, we propose to use the uncertainty of the proximity prediction to prevent the policy from exploiting over-optimistic proximity predictions and yielding undesired behaviors.

6.3.1 Preliminaries

We formulate our problem as a Markov decision process [291] defined through a tuple $(\mathcal{S}, \mathcal{A}, R, P, \rho_0, \gamma)$ of the state space \mathcal{S} , action space \mathcal{A} , reward function $R(s, a, s')$, transition distribution $P(s'|s, a)$, initial state distribution ρ_0 , and discounting factor γ . We define a policy $\pi(a|s)$ that maps from a state s to an action a and correspondingly moves an agent to a new state s' according to the transition probability $P(s'|s, a)$. The policy is trained to maximize the sum of discounted rewards, $\mathbb{E}_{(s_0, a_0, \dots, s_{T_i}) \sim \pi} \left[\sum_{t=0}^{T_i-1} \gamma^t R(s_t, a_t, s_{t+1}) \right]$, where T_i is the variable episode length.

In imitation learning, the learner receives a set of N expert demonstrations, $\mathcal{D}^e = \{\tau_1^e, \dots, \tau_N^e\}$. In this paper, we specifically consider the LfO setup where each demonstration τ_i^e is a sequence of states. Moreover,

we assume that goal information is explicitly or implicitly included in the state s , and all demonstrations are successful; therefore, the final state of each trajectory achieves the task goal.

6.3.2 Learning Goal Proximity Function

To effectively leverage expert demonstrations and generalize to new states or new goals, learning a generalizable reward function is essential. In goal-directed tasks, an estimate of how close an agent is to the goal can be utilized as a dense and direct learning signal. Moreover, predicting the *continuous* goal proximity requires understanding the task structure and thus encourages finding more task-relevant features, resulting in better generalization.

Therefore, instead of learning to simply discriminate agent and expert trajectories, we propose to learn a *goal proximity function*, $f : \mathcal{S} \rightarrow [0, 1]$, which predicts *goal proximity* of a state s , which is a discounted value based on the temporal distance to the goal (i.e. inversely proportional to the number of actions required to reach the goal). In this paper, we define goal proximity as the exponentially discounted proximity $f(s_t) = \delta^{(T_i - t)}$, where $\delta \in (0, 1)$ is a discounting factor and T_i is the episode length. Note that the goal proximity function measures the temporal distance, not the spatial distance, between the current and goal states. Therefore, a single proximity value can entail all information about the task, goal, and any roadblocks. There are alternative ways to define goal proximity, such as linearly discounted proximity [160] and ranking-based proximity [34, 36]. But, in this paper, we use the exponentially discounted proximity as it performs better across most tasks (see appendix, Figure 6.8).

We train a goal proximity function f_ϕ parameterized by ϕ to minimize the following objective:

$$\mathcal{L}_\phi = \mathbb{E}_{\tau_i^e \sim \mathcal{D}^e, s_t \sim \tau_i^e} [f_\phi(s_t) - \delta^{(T_i - t)}]^2. \quad (6.1)$$

Since the goal proximity function trained only on expert demonstrations can overfit to the data, we further train the goal proximity function with online agent experience by setting the target proximity of states in agent trajectories to 0, similar to adversarial imitation learning methods [114]:

$$\mathcal{L}_\phi = \mathbb{E}_{\tau_i^e \sim \mathcal{D}^e, s_t \sim \tau_i^e} [f_\phi(s_t) - \delta^{(T_i-t)}]^2 + \mathbb{E}_{\tau \sim \pi_\theta, s_t \sim \tau} [f_\phi(s_t)]^2. \quad (6.2)$$

By learning to predict the goal proximity, f_ϕ not only learns to discriminate agent and expert trajectories (i.e. predict 0 proximity for an agent trajectory and positive proximity for an expert trajectory) but also acquires the task information about temporal progress entailed in the trajectories. From this freely available additional supervision, the proximity function is required to learn task-relevant features. Hence, the resulting proximity function generalizes better to unseen states and provides more informative learning signals to the policy as empirically shown in Section 6.4.

Due to the lack of environment reward, *successful* agent experience is also used as negative examples for proximity function training, and thus the proximity function learns to predict low goal proximity even for successful trajectories. However, early stopping and learning rate decay can ease this problem [355], and the optimal proximity function still outputs the average of expert and agent labels, which is $\delta^{(T_i-t)}/2$ for ours and 0.5 for GAIL [88].

6.3.3 Training Policy with Proximity Reward

In a goal-directed task, a policy π_θ aims to get close to and eventually reach the goal. We can formalize this objective as maximizing the difference-based *proximity reward* R_ϕ , the increase in goal proximity, at every timestep, which corresponds to making consistent progress towards the goal:

$$R_\phi(s_t, s_{t+1}) = f_\phi(s_{t+1}) - f_\phi(s_t). \quad (6.3)$$

Given the proximity reward R_ϕ , the policy is trained to maximize the expected discounted return:

$$\mathbb{E}_{(s_0, \dots, s_{T_i}) \sim \pi_\theta} \left[\sum_{t=0}^{T_i-1} \gamma^t R_\phi(s_t, s_{t+1}) \right]. \quad (6.4)$$

However, a policy trained with the proximity reward can sometimes acquire undesired behaviors by exploiting over-optimistic proximity predictions on states not seen in the expert demonstrations. This becomes critical when the expert demonstrations are limited and cannot sufficiently cover the state space. To avoid inaccurate predictions leading an agent to undesired states, we propose to (1) fine-tune the proximity function with online agent experience to reduce optimistic proximity predictions; and (2) penalize agent trajectories with high uncertainty in goal proximity prediction.

To alleviate the effect of inaccurate proximity estimation in policy training, we discourage the policy from visiting states with uncertain proximity estimates. Specifically, we model the uncertainty $U_\phi(s_t)$ as the disagreement of an ensemble of proximity functions by computing the standard deviation of their outputs [218, 152]. Then, we use this estimated uncertainty to penalize exploration of states with high uncertainty. The proximity estimate $f_\phi(s_t)$ is the average prediction of the ensemble. With the uncertainty penalty, the modified proximity reward can be written as:

$$R_\phi(s_t, s_{t+1}) = f_\phi(s_{t+1}) - f_\phi(s_t) - \lambda \cdot U_\phi(s_{t+1}), \quad (6.5)$$

where λ is a tunable hyperparameter to balance the proximity reward and uncertainty penalty. A larger λ results in more conservative exploration outside the states covered by the expert demonstrations.

In summary, we propose to learn a goal proximity function to robustly provide a reward signal on states or goals not covered by demonstrations. We train the goal proximity function to estimate how close the current state is to the goal, and train a policy to maximize the goal proximity while avoiding states

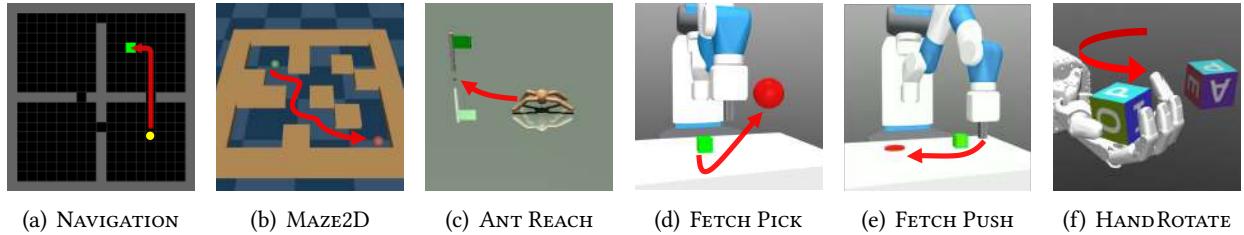


Figure 6.2: Six goal-directed tasks are used for our experiments. (a) The agent must navigate across rooms to reach the goal. (b) The agent needs to navigate the maze to reach the goal. (c) The ant agent must walk towards the flag. (d, e) The robotic arm is required to pick up or push the block towards the goal (red). (f) The dexterous robot hand needs to rotate the block in-hand to the desired rotation.

with uncertain proximity predictions. We jointly train the proximity function and policy as described in appendix, Algorithm 2.

6.4 Experiments

In this paper, we propose a generalizable LfO algorithm that leverages task progress information (i.e. goal proximity) freely acquired from demonstrations. Hence, in our experiments, we aim to answer the following questions: (1) Does our method lead to policies that generalize better to states and goals not in the demonstrations? (2) How does our method’s efficiency and performance compare against prior work in LfO and LfD? (3) What factors contribute to the performance of our method? To answer these questions we consider diverse goal-directed tasks: navigation, locomotion, and robotic manipulation.

6.4.1 Experimental Setup

To demonstrate the improved generalization capabilities of policies trained with the goal proximity, we benchmark our method under two different setups: expert demonstrations are collected from (1) only a fraction of the possible initial and goal states (e.g. 25%, 50% coverage) and (2) initial states with smaller amounts of noise. These generalization experimental setups serve to mimic the reality that expert demonstrations may be collected in a different setting from agent learning. For instance, due to the cost of

demonstration collection, the demonstrations may poorly cover the state space, which corresponds to the setup (1). Likewise, in the setup (2), demonstrations may be collected in controlled circumstances with little noise. Then, an agent in an actual environment would encounter more noise than presented in the demonstrations, leading to a wider initial state distribution.

In our experiments, we use the discounting factor $\delta = 0.95$ for the goal proximity. We use an ensemble of 5 proximity functions to model uncertainty across all tasks. For policy optimization, we use PPO [265], which is widely used in LfO and LfD methods, and its hyperparameters are tuned for each method and task (see appendix, Table 6.2). Each baseline implementation is verified against the results reported in its original paper. We train each task with 5 random seeds and report mean and standard deviation. See Section 6.6.6 for further implementation details.

6.4.2 Baselines

We compare our method to the state-of-the-art methods in LfO (BCO, GAIfO, GAIfO-s) as well as LfO with reward (GoalGAIL) and LfD (BC, GAIL, SQIL) approaches, which require additional supervisions, such as task reward and expert actions:

- **BCO** [300] learns an inverse dynamics model from environment interaction to provide action labels in demonstrations for behavioral cloning.
- **GAIfO** [301] trains a discriminator with state transitions (s, s') , instead of (s, a) as in GAIL.
- **GAIfO-s** [335] learns a discriminator based off a single state, not a state transition as with GAIfO.
- **GoalGAIL** [67] uses goal reaching reward and relabeling to improve sample efficiency of GAIL.
- **BC** [236] fits a policy to the demonstration state-action pairs (s, a) with supervised learning.
- **GAIL** [114] is an adversarial imitation learning with a discriminator trained on state-action pairs (s, a) from both expert and agent.

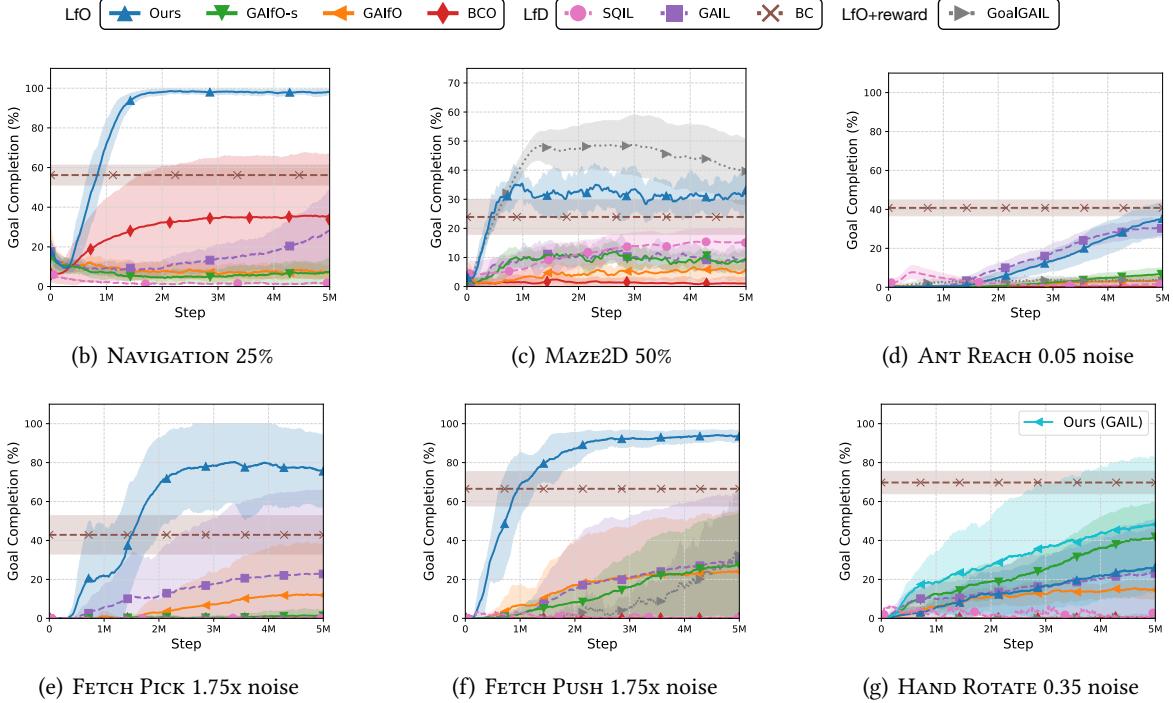


Figure 6.3: Goal completion rates of our method and baselines. The agent must generalize to a wider state and goal distributions than seen in the demonstrations. Demonstrations cover only a part of states (a, b) or are generated with less noise (c, d, e, f). Our method learns more stably, faster, and achieves higher goal completion rates than prior LfO methods. Moreover, our method outperforms the LfD baselines in NAVIGATION, FETCH tasks, and MAZE2D, and achieves comparable results in ANT REACH. GoalGAIL performs well in MAZE2D since it can easily acquire environment rewards.

- **SQIL** [246] is a sample-efficient imitation learning method which adds expert transitions (s, a) with reward 1 to the replay buffer of off-policy RL and assigns 0 reward to all agent experience.

6.4.3 Navigation

We first examine the NAVIGATION task between four rooms shown in Figure 6.2(a) to demonstrate generalization capability of our method, and visualize the learned goal proximity function. The agent observes the $19 \times 19 \times 4$ 2D map of the maze and moves in one of four directions. In this task, the agent starting and goal positions are randomly sampled (see an example in appendix, Figure 6.12). We provide 250 expert demonstrations obtained using a shortest path algorithm. During demonstration collection, we hold out

0%, 25%, 50%, and 75% of the possible agent starting and goal positions uniformly at random. In contrast, during agent learning and evaluation, start and goal positions are sampled from all possible positions.

Figure 6.3(b) shows that our method achieves near 100% success rate in 2M environment steps even with demonstrations only covering 25% of starting and goal states, while other LfO methods fail to learn the task. Although BC, GAIL, and BCO achieve success rates of about 60%, 30%, and 35%, respectively, they show limited generalization to unseen configurations. This result shows that the learned goal proximity function generalizes well to unseen configurations.

Figure 6.4(e) visualizes the proximity function trained with 50% coverage demonstrations and 250k steps of agent training. Our proximity function predicts high proximity near the goal and lower proximity when the agent is farther away from the goal. This demonstrates that our proximity function can learn the semantic, non-euclidean relationship between high-dimensional observations and goals. Since the proximity function is conditioned on the state, similar states are likely to have similar predicted proximity, and thus the proximity function learns a spatially consistent measure of proximity from temporal supervision. Moreover, as the task progress is a relative position within a trajectory, both slow and fast demonstrations result in the same task progress. More visualizations can be found in appendix, Section 6.6.5.

Finally, we investigate our hypothesis that the goal proximity function allows for greater generalization, which results in better performance with smaller demonstration coverages. We compare the cases where extreme (25% coverage), moderate (50% and 75% coverage), and no generalization (100% coverage) are required. Figure 6.3(b) and Figure 6.4 show that our method consistently achieves almost 100% success rates in 2M steps across all coverages and is not as affected by the increasingly difficult generalization settings as baselines. In contrast, all LfO baselines struggle to learn the task when the demonstrations do not cover all configurations. LfD methods also shows limited generalization in 25% coverage since the discriminator can easily learn spurious associations between the actions and labels, which hurts generalization to new

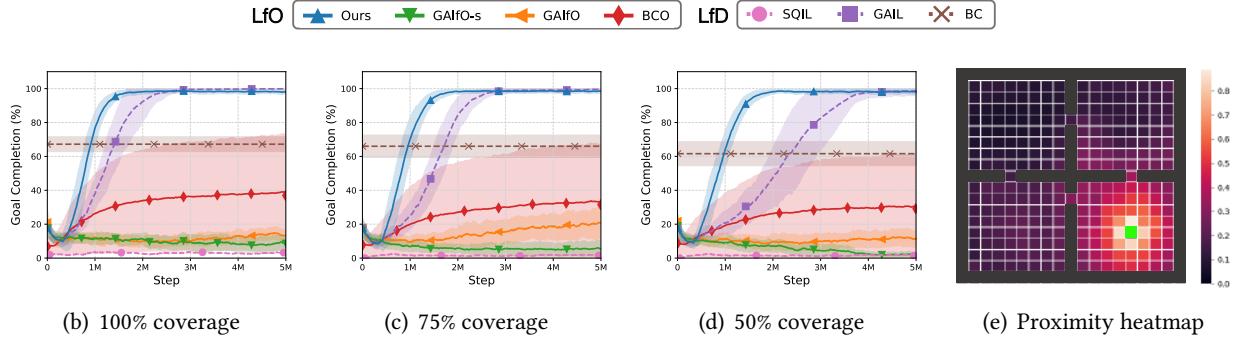


Figure 6.4: Analyzing the effect of improved generalization as the cause for performance increase in our method. (a) Performance with no generalization required. (b, c) Performance with increasing difference between start and goal distributions of demonstrations and agent learning. (d) Visualization of the learned proximity function for a fixed goal (green) in the 50% coverage case. The proximity function was evaluated for every state on the grid; lighter cells correspond to states with higher estimated proximity to the goal.

actions. This supports our hypothesis that the goal proximity function is able to capture the task structure and therefore, generalize better to unseen configurations.

6.4.4 Maze2D

We further evaluate our method in MAZE2D [87] with the medium maze, a continuous version of NAVIGATION. The agent observes its position, velocity, and goal position, and then outputs an x- and y-velocity to navigate the maze. The agent starting and goal positions are randomly sampled. We collect 100 demonstrations (18k transitions) using a planner from Fu et al. [87].

Our method outperforms LfO baselines over all demonstration coverages (see appendix, Figure 6.7). More importantly, in the low coverage case, our method outperforms BC, which has access to expert actions, as shown in Figure 6.3(c). This could be because our proximity function generalizes well whereas BC is not robust to unseen states under small demonstration coverages. On the other hand, GoalGAIL shows the best performance regardless of coverages as the task can be easily solved with the sparse reward and goal relabeling, which is not available for our method and other baselines.

6.4.5 Ant Locomotion

In ANT REACH [91], the quadruped ant is tasked to reach a randomly generated goal, which is along the perimeter of a half circle of radius 5 m centered around the ant (see Figure 6.2(c)). The 132D state consists of joint angle, velocity, contact force, and the goal position relative to the agent. We collect 1k demonstrations (25k transitions) using the pre-trained policy (trained for 40M steps). When demonstrations are collected, no noise is added to the initial pose of the ant whereas random noise is added to the initial pose during policy learning, which requires the reward functions to generalize to unseen states.

In Figure 6.3(d), with 0.05 added noise, our method achieves 35% success rate while BCO, GAIIfO, and GAIIfO-s achieve 1%, 2%, and 7%, respectively. This result illustrates the importance of learning proximity as opposed to discriminating expert and agent states for generalization to unseen states. The performance of GAIIfO and GAIIfO-s drops drastically with larger joint angle randomness as shown in appendix, Figure 6.7. As ANT REACH is not as sensitive to noise in actions compared to other tasks, BC and GAIL show superior results but our method still achieves comparable performance.

6.4.6 Robotic Manipulation

We evaluate our method in two robotic manipulation tasks with the 7-DoF Fetch robotics arm: FETCH PICK and FETCH PUSH [235]. The robot must grasp and move a block to a target position for FETCH PICK, and push a block to a target position for FETCH PUSH. The 16D state consists of the gripper pose, object pose, the gripper pose relative to the object, and goal position. Both the initial and target positions of the block are randomly initialized. We generate 1k demonstrations using a hard-coded policy, consisting of 33k and 28k transitions for FETCH PICK and FETCH PUSH, respectively. The policy is trained in an environment with larger noise applied to the starting and target block positions.

In FETCH PICK, our method achieves about 80% success rate outperforming all baselines, despite LfD methods learning with expert actions (see Figure 6.3(e)). The best performing baseline BC only obtains

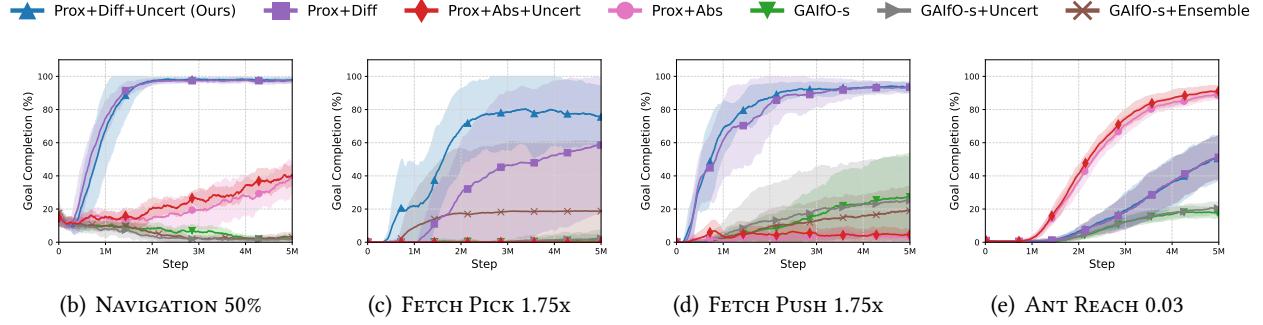


Figure 6.5: Analysis of the contribution of goal proximity function, uncertainty penalty, and reward formulation to the performance. “Prox” uses the goal proximity function while “GAIfO-s” does not. “+Diff” uses $R(s_t, s_{t+1}) = f(s_{t+1}) - f(s_t)$ and “+Abs” uses $R(s_t) = f(s_t)$ as a reward. “+Uncert” adds the uncertainty penalty to the reward. “+Ensemble” uses an ensemble for the discriminator.

around 40% success rate. The high variance in performance between seeds comes from the difficulty of learning the grasping behavior with large noise. In **FETCH PUSH**, our method outperforms baselines in generalization to unseen states by achieving more than 90% success rate (see Figure 6.3(f)). This shows that our proximity function is able to accelerate policy learning in continuous control environments with superior generalization capability.

6.4.7 Dexterous Hand Manipulation

We evaluate our method in a challenging in-hand object manipulation task [235], **HAND ROTATE** as shown in Figure 6.2(f). In **HAND ROTATE**, a 24-DoF Shadow Dexterous Hand must in-hand rotate a block to a target z -axis rotation. The state consists of the agent’s joint angles and velocities, object pose, and the target rotation. Due to the high dimension of the state (68D) and action space (20D), **HAND ROTATE** is extremely challenging for both RL and IL without dense reward. We therefore ease the task by constraining the possible initial and target z rotations to $[-\frac{\pi}{32}, \frac{\pi}{32}]$ and $[\frac{\pi}{3}, \frac{\pi}{2}]$. We collect 10k demonstrations (98k transitions) using a pre-trained policy (trained for 8M steps).

In Figure 6.3(g), GAIfO-s performs well because its reward function is biased to provide large negative rewards encouraging the agent to end the episode early which is only possible by succeeding. In contrast,

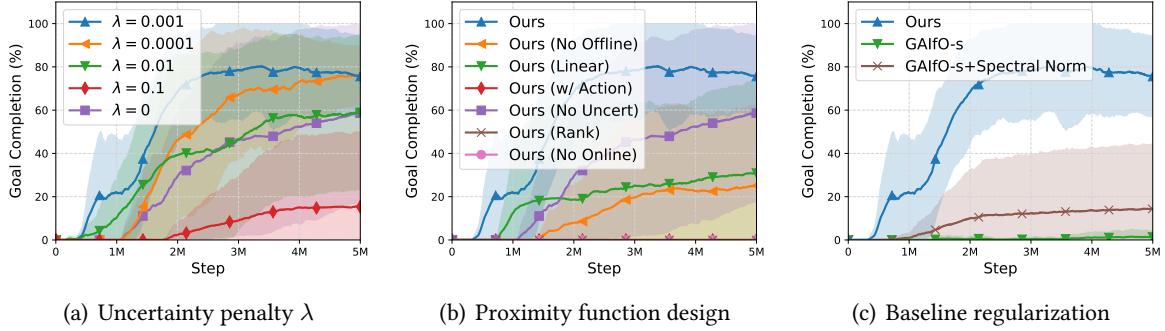


Figure 6.6: Ablation of our method and comparison to a regularized baseline on FETCH PICK 1.75x to investigate (a) effects of uncertainty penalty coefficient λ ; (b) effects of proximity function design and online/offline training; and (c) generalization capability of a regularization technique and our proximity function.

our difference-based reward is designed to provide positive rewards, which does not exploit this task property, and performs poorly even with an additional constant penalty -0.005 every step. To test the generalization capability of our proximity function, we additionally examine a variant of our method (Ours-GAIL), which uses the same reward formulation as GAIIfO-s, $\log f_\phi(s_t) - \log(1 - f_\phi(s_t))$. With this biased reward function, our method outperforms both GAIIfO-s and GAIL, which verifies the benefit of our proximity function in generalization to noisy environments. While BC achieves the high success rate with 10x more demonstrations compared to other tasks, SQIL shows poor performance due to the lack of the negative reward bias.

6.4.8 Ablation Study

Dissecting proximity reward We analyze the contribution of the proximity function, reward formulation, and uncertainty penalty to our method’s performance across four tasks in Figure 6.5. Adding uncertainty to GAIIfO-s (GAIIfO-s+Uncert) produced a 18.4% boost in average success rate compared to regular GAIIfO-s. Proximity supervision, without the uncertainty penalty, resulted in a 66.7% increase in average performance over GAIIfO-s with the difference-based reward $f(s_{t+1}) - f(s_t)$ (Prox+Diff) and 25.8%

with the absolute reward $f(s_t)$ (Prox+Abs). This higher performance means **modeling proximity is more important than the uncertainty penalty** for our method.

Although we choose difference-based reward with exponentially discounted goal proximity, the goal proximity can be either linear or exponential discounting, and both can be used for either a difference-based or absolute reward, which perform differently across tasks. For example, the difference-based proximity reward is better for policy learning than the absolute proximity reward except on ANT REACH and HAND ROTATE, where the bias of the absolute reward [145] helps the agent survive longer and reach the goal. This is a fundamental problem in IRL, where inductive bias in reward functions is crucial and varies by tasks [145]. Nonetheless, our extensive experiments (Figure 6.5, 6.6(b), 6.8) show that our goal proximity reward provides a more stable and generalizable learning signal than baselines under the same reward bias.

Moreover, we found that **the uncertainty penalty and proximity function have a synergistic interaction**. Combining both the proximity and uncertainty gives a 68.7% increase with the difference-based reward (Prox+Diff+Uncert) and 26.4% increase with the absolute reward (Prox+Abs+Uncert). The uncertainty penalty is especially important for the proximity function as it models fine-grain temporal information where inaccuracies can be easily exploited, as opposed to the binary classification of other adversarial imitation learning methods.

Ensemble networks Next, we study if the robustness of our method comes from the use of ensemble networks or task progress. We verify this by applying ensemble of discriminators to the best performing baseline, GAIfO-s. Figure 6.5 shows that GAIfO-s with ensemble networks (GAIfO-s+Ensemble) only achieves 19.6% higher success rates, but this is still 39.7% lower than our method on average. Therefore, **the use of task progress is key** to learn a generalizable reward, not the use of ensemble networks.

Regularization of discriminators In our experiments above, we show that our goal proximity function is generalizable to unseen states and goals, which leads to successful imitation learning. We verify

whether standard regularization techniques, such as spectral normalization [197], can also provide the same generalization benefit. In the FETCH PICK 1.75x noise setting (Figure 6.6(c)), GAIfO-s without regularization struggles to learn, achieving only a 1.43% success rate. Not surprisingly, applying spectral normalization [197] to the discriminator of GAIfO-s improves the success rate to 14.56%, which suggests that generalization of the reward function is key to imitation learning with insufficient demonstration coverage. Despite this improvement, our method performs much better at 75.45% success. In summary, **predicting goal proximity enables significantly better generalization than regularization** on the baselines. Figure 6.10 in appendix show similar results across most other tasks.

Uncertainty penalty coefficient λ In Figure 6.6(a), we investigate how the uncertainty penalty coefficient λ affects the performance, showing that our method performs the best with $\lambda = 0.001$. A higher or lower λ yields worse performance since a higher λ prevents exploring unseen states while a lower λ encourages exploiting uncertain predictions.

Proximity function training In Figure 6.6(b), we test the importance of online and offline training of the proximity function. Note that we update the policy with online interactions in both scenarios. The result shows that **online proximity function update is crucial** for our method as the agent fails without online update. Meanwhile, no pre-training, Ours (No Offline), slows down training. Similar results can be observed across all tasks (see appendix, Figure 6.8).

Our ablation experiments show that (1) goal proximity generalizes better and is more informative for policy learning; (2) the difference-based proximity reward generally performs better than the absolute one; and (3) the uncertainty penalty boosts the performance of our method. In conclusion, all three components of proximity, difference-based reward, and uncertainty are crucial for our method.

6.5 Conclusion

We propose a generalizable learning from observation (LfO) method inspired by how humans acquire generalizable task information and learn skills in new situations by watching others performing goal-directed tasks. We specifically propose to use task progress, which is intuitive and readily available task information that can guide an agent closer to the goal. Inspired by this insight, we learn a goal proximity function and utilize it as a dense reward for policy learning. We hypothesize that predicting the task progress requires more task-relevant information than estimating an occupancy measure [114], and thus generalizes to states not seen in the demonstrations. Our extensive experiments on navigation, locomotion, and robotic manipulation show that our goal proximity function improves generalization in imitation learning, which results in better performance compared to LfO methods and comparable performance with LfD methods which learn from expert actions.

In imitation learning, the *generalization* ability can include generalization to (1) unseen states and goals, (2) new visual environments (e.g. background), (3) unseen objects, and (4) different embodiments (e.g. humans to robots or different dynamics). In this paper, we focus on generalization to (1) unseen states and goals. This is especially important in imitation learning when the number of demonstrations is not sufficient to cover all possible states and goals. This is very common in imitation learning due to costly demonstration collection. Our approach suggests an effective way of using the demonstrations with limited coverage by learning the generalizable goal proximity reward.

Generalization to a different environment and embodiment is another important research direction and this is indeed our immediate future work. Recent advances in generalizable representation learning [267, 282, 321], robust policy learning [125, 149], and cross-domain correspondence [347] enable us to train a policy that generalizes to new environments and embodiments. Yet, these approaches are orthogonal and complementary to our method as our goal proximity function can be trained on top of the learned representations [267, 282, 321, 125, 149, 347]. We believe that our method can be combined with these

approaches and improve their performance with better demonstration efficiency and additional supervision about task progress.

Societal Impact Our method aims to increase the ability of autonomous agents, such as robots and self-driving cars, to imitate experts (e.g. humans) from observation alone. This enables autonomous agents to utilize data even without expert actions, such as kinesthetic demonstrations and video demonstrations. Ultimately, it could allow autonomous agents to acquire skills even from watching Youtube videos. Since our method learns from experts, it inherits any biases of the demonstrator, such as sub-optimal or unsafe behaviors. Additionally, demonstrations are an easy and intuitive way to specify behaviors, its potential for automation is a threat to job security. However, we overall see enormous benefit with this technology increasing human quality of life and automating difficult jobs.

6.6 Appendix

6.6.1 Comparison with GAIL and Its Variants

Our method shares a similar adversarial training process with GAIL [114]. First of all, similar to the discriminator in GAIfO-s [335], our proximity function takes only the current state as input. However, rather than training the discriminator to classify expert from agent, we train the proximity function to regress to the proximity labels which are 0 for agent and the time discounted value between 0 and 1 for expert. Our reward formulation also differs from GAIL approaches which gives a log probability reward based on the discriminator output. We instead incorporate a proximity estimation uncertainty penalty and a difference-based proximity reward as shown in Equation 6.3.

6.6.2 Failure of GAIIfO and SQIL

We found that SQIL training is unstable and often collapses after some amount of training steps (see Ant experiments in Figure 6.7). Similar trends can be observed in the original paper [246] and other recent papers [292, 240]. We hypothesize that GAIIfO easily overfits to demonstrations compared to other baselines (e.g. GAIIfO-s) since GAIIfO conditions its discriminator on both the current and next observations.

We evaluated these methods with the demonstrations from the same initial and goal state distributions in the first column of Figure 6.7. Even though they are trained for the same goal distributions as the demonstrations, they still overfit to the demonstration states and thus cannot generalize to unseen states encountered during online rollouts for most tasks.

6.6.3 Analysis on Generalization of Our Method and Baselines

By learning to predict the goal proximity, the proximity function not only learns to discriminate expert and agent states but also models task progress, which encourages acquiring task-relevant information. With this additional supervision on learning goal proximity, we expect the proximity function to provide a more informative learning signal to the policy and generalize better to unseen states than baselines which easily overfit the reward function to expert demonstrations. To analyze how well our method and the baselines can generalize to unseen states, we vary the difference between the states encountered in expert demonstrations and agent training as described in Section 6.4.

One way we vary the difference between expert demonstrations and agent learning is restricting the expert demonstrations to only cover parts of the state space. For NAVIGATION and MAZE2D, we show results for expert demonstrations that cover 100%, 75%, 50%, and 25% of the state space. For the discrete state space in NAVIGATION, we restrict expert demonstrations to the fraction of possible agent start and goal configurations. For MAZE2D, we break the maze into 6×6 cells and sample a part of the cells for starting states and another part for goal states.

Likewise, we also measure generalization by adding more noise to the initial state during agent learning. On `FETCH PICK`, `FETCH PUSH`, `ANT REACH`, and `HAND ROTATE` we show results for four different noise settings. For the two `FETCH` tasks, the 2D sampling region of the object and goal is scaled by the noise factor. For `ANT REACH`, uniform noise scaled by the noise factor is added to the initial joint angles, whereas the demonstrations have no noise. For `HAND ROTATE`, uniform noise scaled by the noise factor is added to the possible initial and target object pose. If our method allows for greater generalization from the expert demonstrations, our method should perform well even under states different than those in the expert demonstrations.

The results of our method and baselines across varying degrees of generalization are shown in Figure 6.7. Note that the results in the main paper are for 1.75x noise in `FETCH PICK` and `FETCH PUSH`, 0.05 noise in `ANT REACH`, 0.35 noise in `HAND REACH`, and 25% coverage in `Maze2D`. Across both harder and easier generalization, our method demonstrates more consistent performance compared to baseline methods. While GAIfO-s performs well on high coverage or low noise, which require little generalization in agent learning, its performance deteriorates as the expert demonstration coverage decreases.

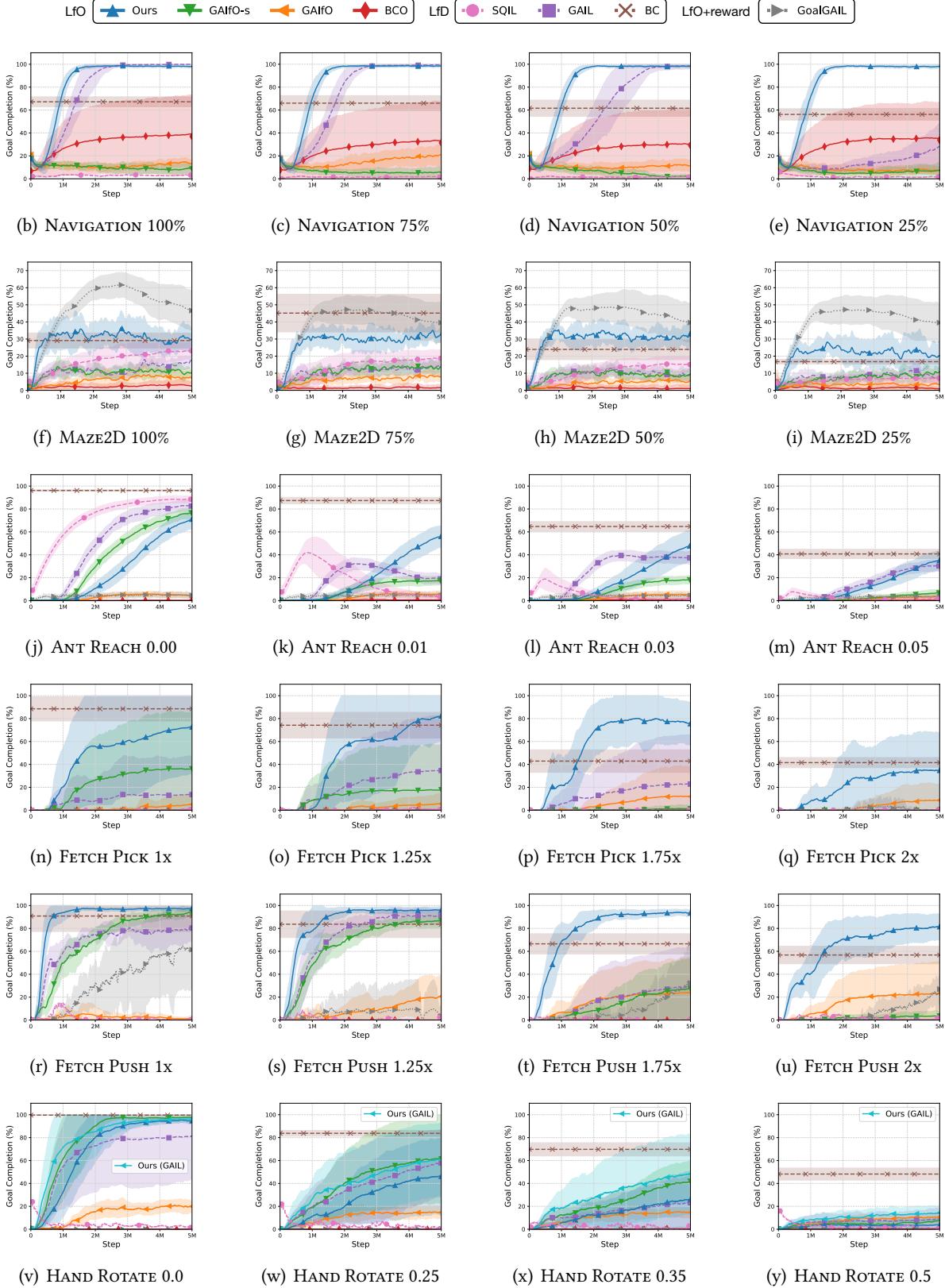


Figure 6.7: Analyzing generalization to unseen states from expert demonstrations. NAVIGATION and MAZE2D tasks are tested with different coverages of state spaces in demonstrations, while FETCH, ANT REACH, and HAND ROTATE tasks are tested in more noisy environments. The number indicates the amount of additional noise in agent learning compared to that in the expert demonstrations, with more noise requiring harder generalization. The noise level increases from left to right.

6.6.4 Further Ablations

We include additional ablations to further highlight the advantages of our main proposed method over its variants. We evaluate against the same ablations proposed in the main paper (Figure 6.6(b)), but across all environments. We present all these results in Figure 6.8.

Our method shows the best performance in the majority of environments. In all tasks, incorporating online updates is crucial since the proximity function can overfit to expert trajectories and poorly generalize to agent trajectories. Updating the proximity function with online agent experience lowers the proximity prediction outside of the expert trajectories and thus leads an agent to follow the expert. Our method with the uncertainty penalty shows superior performance in FETCH PICK and HAND ROTATE, while it performs similarly with our method without the uncertainty penalty in other environments. Our method using the linear proximity function achieves similar to or slightly lower performance than the exponential proximity function used in the main paper. Offline pre-training of the proximity function is also helpful in most environments.

We also compare to an ablation which learns the proximity function through a ranking-based loss similar to Brown et al. [34] and Burke et al. [36]. However, we empirically found it to be ineffective and difficult to train. This ranking-based loss uses the criterion that for two states from an expert trajectory s_{t_1}, s_{t_2} , the proximities should obey $f(s_{t_1}) < f(s_{t_2})$ if $t_1 < t_2$. We therefore train the proximity function with the cross entropy loss $-\sum_{t_i < t_j} \log \frac{\exp f_\phi(s_{t_j})}{\exp f_\phi(s_{t_i}) + \exp f_\phi(s_{t_j})}$. We incorporate agent experience by adding an additional loss which ranks expert states above agent states for randomly sampled pairs of expert and agent states (s_e, s_a) through the cross-entropy loss:

$$- \sum_{s_a \sim D^e, s_e \sim \pi_\theta} \log \frac{\exp f_\phi(s_e)}{\exp f_\phi(s_a) + \exp f_\phi(s_e)}. \quad (6.6)$$

Unlike the discounting factor in the discounting-based proximity function, the ranking-based training requires no hyperparameters. However, as shown in Figure 6.8, the lack of supervision on ground truth proximity scores results in less meaningful predicted proximity and a worse learning signal for the agent, which could explain its poor performance.

We also show results for applying spectral normalization [197] to GAIIfO-s [301] in Figure 6.10 across all tasks. While regularizing the GAIIfO-s discriminator can consistently improve its performance, it still cannot generalize as well as our method for the majority of tasks. As mentioned in Section 6.4.7, GAIIfO-s has a bias to provide negative rewards encouraging the agent to end the episode early, which is a desirable property for the HAND ROTATE task. Vanilla GAIIfO-s therefore performs better than our method in this environment, and spectral normalization for the discriminator further improves GAIIfO-s performance.

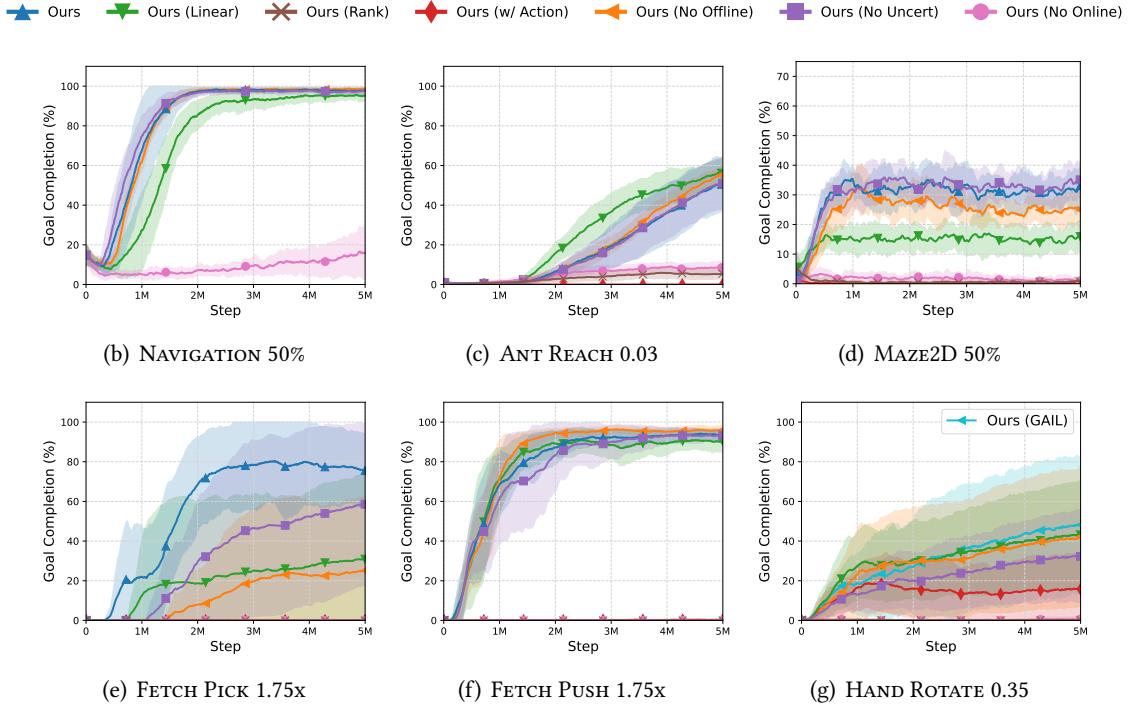


Figure 6.8: Ablation on proximity function design and online/offline proximity function training. We compare our method to the proximity function with actions as input or with a ranking-based objective (Equation 6.6). Our method shows consistently superior or comparable performance over all ablations.

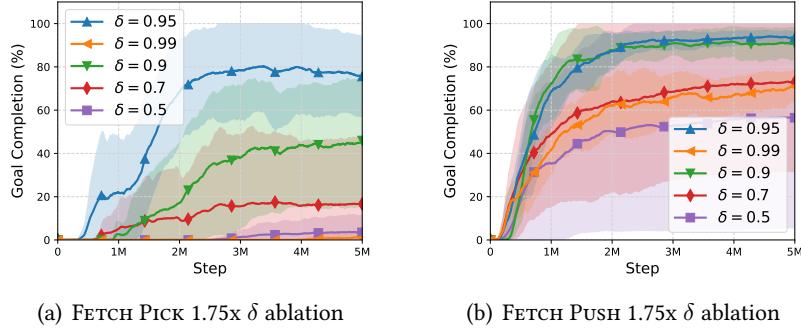


Figure 6.9: Analyzing different choices of the proximity discounting factor δ for training the proximity function. The model learns similarly well over a range of δ values around 0.95, but struggles for too large or too small δ .

6.6.5 Qualitative Results

It is important for agent learning that the proximity function gives higher values for states that are temporally closer to the goal. To verify this intuition, we visualize the proximity values predicted by the proximity function in a successful episode from agent learning in Figure 6.11. In Figure 6.11, we can observe that the predicted proximity increases as the agent moves closer to the goal (except HAND ROTATE). This provides an example of the proximity function generalizing to agent experience and providing a meaningful reward signal for agent learning.

We notice that while the predictions increase as the agent nears the goal, the proximity prediction values are often low (<0.1) as shown in Figure 6.11(c). These low values are mostly predicted for the states not covered in the demonstrations due to the adversarial online training of the proximity function. During online proximity function training, we label agent experience with 0 proximity and therefore proximity predictions get lower, especially for states not in the demonstrations.

For HAND ROTATE, the proximity function fails to predict increasing proximity for states near the goal as an agent cannot learn to imitate the exact expert trajectories. Instead, due to the negatively biased reward, the agent finds a new way to solve the task as discussed in Section 6.4.7 and therefore achieves low proximity predictions even for successful trajectories as shown in Figure 6.11(e). However, our method still

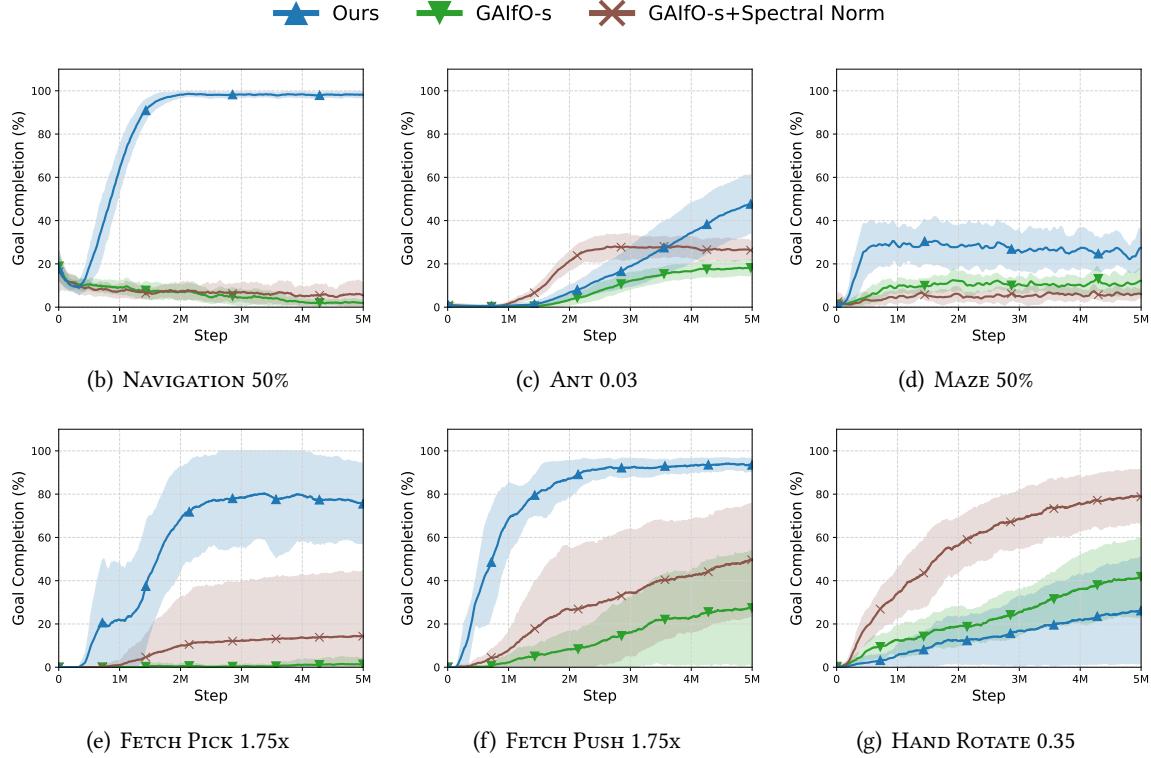


Figure 6.10: Effect of applying spectral normalization to the GAIfO-s baseline compared to the performance of our method. While regularization helps GAIfO-s, it still is outperformed by our method in the majority of tasks.

provides relatively higher proximity values near goal states compared to baseline methods, which leads our agent to achieve higher performance in noisy environments.

6.6.6 Implementation Details

We use PyTorch [223] for our implementation and all experiments are conducted on a workstation with an Intel Xeon E5-2640 v4 CPU and a NVIDIA Titan Xp GPU. Most adversarial imitation learning methods and our method are trained for around 3 hours with 32 parallel workers. GoalGAIL and SQIL training takes around 48 hours since they use off-policy optimization with a single worker.

6.6.6.1 Environment Details

In this section, we summarize details of the six goal-directed tasks discussed in this paper. For all environments, the starting and goal states are randomly initialized. All units in this section are in meters and radians unless otherwise specified. The summary of observation spaces, action spaces, and episode lengths are described in Table 6.1. To evaluate the generalization capability of our method and baselines, we constrain the coverage of expert demonstrations or add additional starting state noise during agent learning as discussed in Section 6.6.3.

NAVIGATION [50] In NAVIGATION, the state consists of a one-hot vector for each grid cell encoding wall, empty space, agent, or goal. NAVIGATION has four discrete actions for moving in four directions. We collect 250 expert demonstrations using the shortest path algorithm, BFS search. The 25% holdout region is visualized in Figure 6.12.

MAZE2D [87] In MAZE2D, the state consists of the agent’s 2D position, velocity, and the goal position. The point mass agent moves around the maze by controlling the continuous value of its (x, y) velocity. The only modification in this environment from maze2d-medium-v1 [87] is the episode length reduced from 600 to 400. We collect 100 expert demonstrations using a planner provided by Fu et al. [87].

ANT REACH [91] In ANT REACH the state consists of joint angle, velocity, force and the relative goal position, and the agent is controlled using joint torque control. We collect 1,000 demonstrations using an expert policy trained using PPO [265] based on the reward function $R(s, a) = 1 - 0.2 \cdot \|p_{ant} - p_{goal}\|_2 - 0.005 \cdot \|a\|_2^2$, where p_{ant} and p_{goal} are (x, y) -positions of the ant and goal, respectively, and a is an action. Please refer to the code for more details.

FETCH PICK and FETCH PUSH [235] The actions in the FETCH experiments use 3-D end-effector position control and 1-D continuous control for the gripper (fixed for FETCH PUSH). A 16-dimensional state in FETCH

tasks consists of the relative position of the goal from the object, relative position of the end-effector to the object, and robot joint state. We found that not including the velocity information was beneficial for all learning from observation approaches in **FETCH** tasks. In **FETCH PICK**, we generate 1,000 demonstrations by hard coding the Sawyer Robot to first reach above the object, then reach down and grasp, and finally move to the target position. Similarly, in **FETCH PUSH**, we collect 664 demonstrations by hard coding the Sawyer to reach behind the object and then execute a planar push towards to the goal.

HAND ROTATE [235] The original task `HandManipulateBlockRotateZ-v0` proposed in Plappert et al. [235] is challenging to solve without reward due to its large and combinatorial state space and large action space. Hence, we reduce the initial and goal z rotations of the block to $[-\frac{\pi}{32}, \frac{\pi}{32}]$ and $[\frac{\pi}{3}, \frac{\pi}{2}]$. The 68-D state space consists of the agent’s joint angles and velocities, and object pose. The 20-D action space is for joint torque control of 24-DoF Shadow Dexterous Hand. We collect 10,000 demonstrations using an expert policy trained with DDPG+HER [14] using a sparse reward.

6.6.6.2 Network Architectures

Actor and critic networks: We use the same architecture for actor and critic networks except for the output layer where the actor network outputs an action distribution while the critic network outputs a critic value. For **NAVIGATION**, the actor and critic network consists of $CONV(3, 2, 16) - ReLU - MaxPool(2, 2) - CONV(3, 2, 32) - ReLU - CONV(3, 2, 64)$ followed by two fully-connected layers with hidden layer size 64, where $CONV(k, s, c)$ represents a c -channel convolutional layer with kernel size k and stride s . For other tasks, we model the actor and critic networks as two separate 3-layer MLPs with hidden layer size 256. For the continuous control tasks, the last layer of the actor MLP has two heads to output the mean and standard deviation of a Gaussian distribution which an action is sampled from. We use the ReLU activation for **NAVIGATION** and tanh for other tasks.

Goal proximity function and discriminator: The goal proximity function and discriminator use a CNN encoder (the same CNN architecture as the actor and critic networks) followed by a hidden layer of size 64 for NAVIGATION and a 3-layer MLP with a hidden layer of size 64 for other tasks. When measuring the uncertainty of predictions, we use an ensemble of 5 networks.

6.6.6.3 Training Details

For our method and all baselines except BC [236] and BCO [300], we train policies using PPO [265]. The hyperparameters for policy training are shown in Table 6.2, while the hyperparameters for the proximity and discriminator function are shown in Table 6.3. For our method, we found it helpful to normalize the reward based on the moving average and standard deviation of returns. We also did so for baselines when it helped.

For hyperparameter tuning, we searched over entropy coefficients $\{0.0001, 0.001, 0.01\}$, state normalization $\{\text{True}, \text{False}\}$, uncertainty coefficient $\{0.0001, 0.001, 0.01, 0.1\}$, learning rates $\{0.0001, 0.0003, 0.001\}$, and reward normalization $\{\text{True}, \text{False}\}$.

In BC, the demonstrations were split into 80% training data and 20% validation data. The policy was trained on the training data until the validation loss stopped decreasing. The policy is then evaluated for 1,000 episodes to get an average success rate.

In GAIIfO-s and GAIL, we use the reward form of $\log D(s) - \log(1 - D(s))$ and $\log D(s, a) - \log(1 - D(s, a))$, respectively, from Finn et al. [78] and Fu, Luo, and Levine [88].

For GoalGAIL [67], we use the default hyperparameters used in the original implementation. For the policy network, we use a deterministic policy for DDPG [170] and use the tanh activation to normalize the policy output between $[-1, 1]$. We update the policy and critic every 2 environment steps and the discriminator every 10 environment steps to prevent overfitting.

Algorithm 2 Imitation learning with learned goal proximity

Require: Expert demonstrations $\mathcal{D}^e = \{\tau_1^e, \dots, \tau_N^e\}$

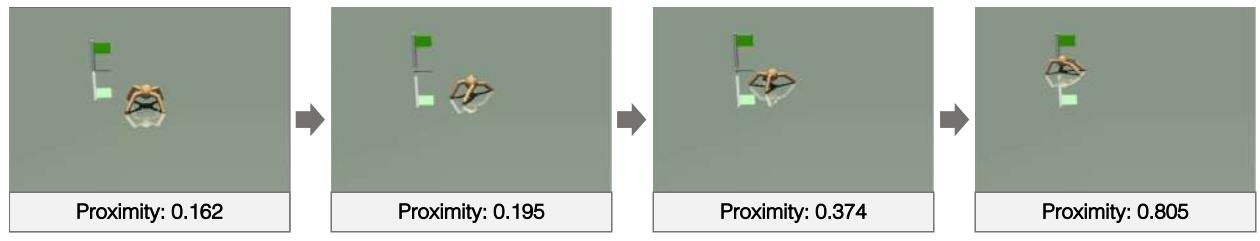
- 1: Initialize goal proximity function f_ϕ and policy π_θ
- 2: **for** $i = 0, 1, \dots, M$ **do**
- 3: Sample expert demonstration $\tau^e \sim \mathcal{D}^e$
- 4: Update f_ϕ with τ^e to minimize Equation 6.1 ▷ Offline proximity function training
- 5: **end for**
- 6: **for** $i = 0, 1, \dots, L$ **do**
- 7: Rollout trajectories $\tau_i = (s_0, \dots, s_{T_i})$ with π_θ
- 8: Compute proximity reward $R_\phi(s_t, s_{t+1})$ for $(s_t, s_{t+1}) \sim \tau_i$ using Equation 6.5 ▷ Policy training
- 9: Update π_θ using any RL algorithm
- 10: Update f_ϕ with τ_i and $\tau^e \sim \mathcal{D}^e$ to minimize Equation 6.2 ▷ Online proximity function training
- 11: **end for**

Table 6.1: Environment details. In NAVIGATION and MAZE2D, the goal and agent are randomly initialized anywhere on the grid. In ANT REACH, the angle of the goal and the velocity of the agent are randomly initialized. The goal and object noise in FETCH describes the amount of uniform noise applied to the (x, y) coordinates of the object and goal. In HAND ROTATE, the state and goal noises are applied to the initial and goal object rotations.

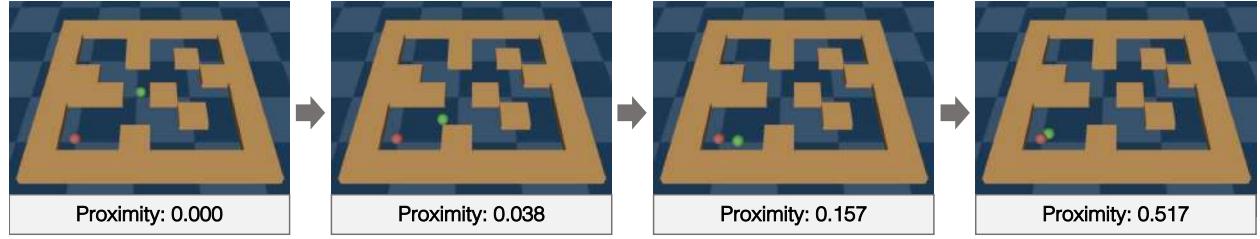
	State	Action	Goal noise	State noise	Episode len.	# demos
NAVIGATION	(19, 19, 4)	4	-	-	50	250
MAZE2D	6	2	-	-	600	100
ANT REACH	132	8	$\theta \in [0, \pi]$	$v \in [\pm.005]$	50	1,000
FETCH PICK	16	4	$(x, y) \in [\pm.02, \pm.05]$		50	1,000
FETCH PUSH	16	3	$(x, y) \in [\pm.02, \pm.05]$		60	664
HAND ROTATE	68	20	$\theta \in [\frac{\pi}{2}, \frac{\pi}{3}]$	$\theta \in [\pm \frac{\pi}{32}]$	50	10,000

Table 6.2: PPO hyperparameters used for baselines and our method.

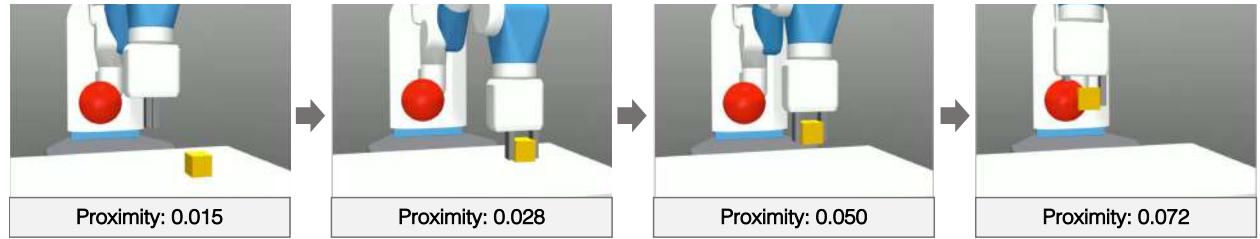
Hyperparameter	Value
Learning Rate	3e-4
Learning Rate Decay	Linear decay
# Mini-batches	4 (NAVIGATION), 32 (others)
# Epochs per Update	4 (NAVIGATION), 10 (others)
Discount Factor γ	0.99
Rollout Size	16,000 (ANT REACH), 4,096 (others)
Entropy Coefficient	0.01 (NAVIGATION), 0.001 (others)
State Normalization	False (NAVIGATION), True (others)



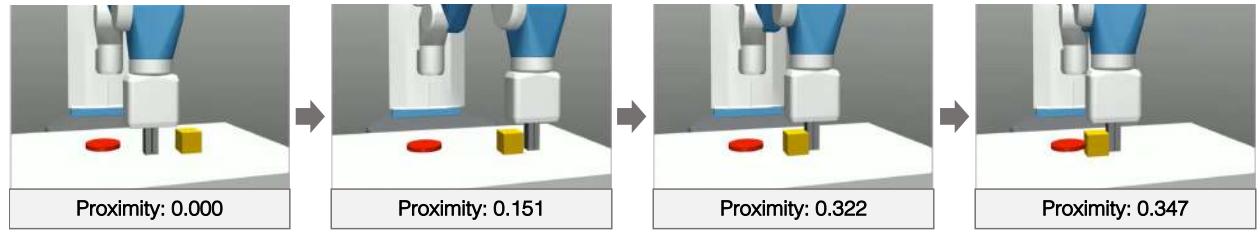
(a) ANT REACH



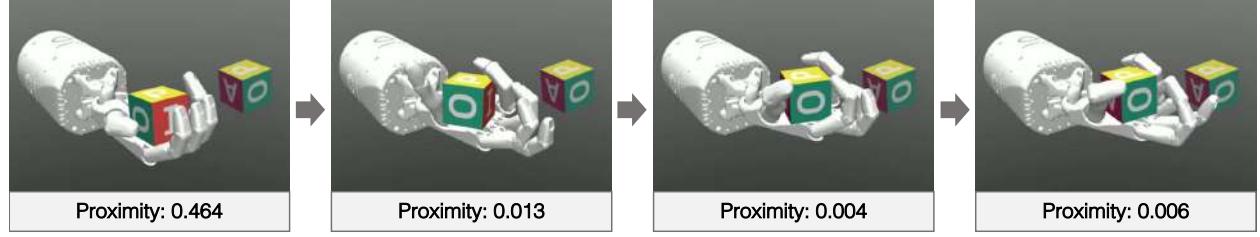
(b) MAZE2D



(c) FETCH PICK



(d) FETCH PUSH



(e) HAND ROTATE

Figure 6.11: Visualizing the proximity predictions for a successful trajectory from agent learning. Four informative frames are selected from the overall trajectory and the predicted proximity value is displayed below. The proximity prediction visualization for NAVIGATION can be found in Figure 6.4(e).

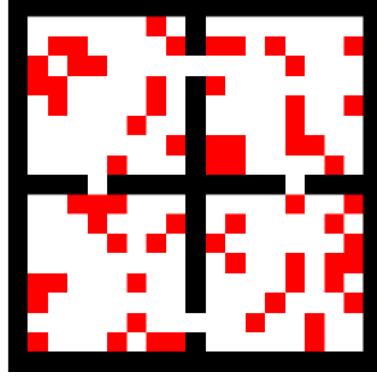


Figure 6.12: The goals of the expert demonstrations in red for the NAVIGATION 25% holdout setting.

Table 6.3: Hyperparameters for goal proximity functions (ours) and discriminators (baselines).

Hyperparameter	Value
# Networks for Ensemble	5
# Epochs for Pre-training	5
Discount Factor δ	0.95 (exponential), $1/H$ (linear)
Uncertainty Coefficient λ	0.001 (FETCH), 0.01 (others)
Learning Rate (ours)	1e-3 (NAVIGATION, FETCH, MAZE2D), 1e-4 (ANT REACH, HAND ROTATE)
Learning Rate (baselines)	1e-4
Batch Size	32 (NAVIGATION), 128 (others)
# Updates per Agent Update	1
Experience Buffer Size	16,000 (ANT REACH), 4,096 (others)
Reward Norm. (ours)	True
Reward Norm. (baselines)	True (FETCH), False (others)

Part IV

Task Execution

Chapter 7

Learning to Execute Programs

7.1 Introduction

Humans are capable of leveraging instructions to accomplish complex tasks. A comprehensive instruction usually comprises a set of descriptions detailing a variety of situations and the corresponding subtasks that are required to be fulfilled. To accomplish a task, we can leverage instructions to estimate the progress, recognize the current state, and perform corresponding actions. For example, to make a gourmet dish, we can follow recipes and procedurally create the desired dish by recognizing what ingredients and tools are missing, what alternatives are available, and what corresponding preparations are required. With sufficient practice, we can improve our ability to perceive (*e.g.* knowing when food is well-cooked) as well as master cooking skills (*e.g.* cutting food into same-sized pieces), and eventually accomplish difficult recipes.

Can machines likewise learn to follow and exploit comprehensive instructions like humans? Utilizing expert demonstrations to instruct agents has been widely studied in [82, 344, 331, 225, 283, 69, 322]. However, demonstrations could be expensive to obtain and are less flexible (*e.g.* altering subtask orders in demonstrations is nontrivial). On the other hand, natural language instructions are flexible and expressive [182, 128, 138, 194, 86, 135, 21]. Yet, language has the caveat of being ambiguous even to humans, due to its lacking of structure as well as unclear coreferences and entities. [11, 216] investigate a hierarchical approach, where the instructions consist of a set of symbolically represented subtasks. Nonetheless, those instructions are

not a function of states (*i.e.* describe a variety of circumstances and the corresponding desired subtasks), which substantially limits their expressiveness.

We propose to utilize *programs*, written in a formal language, as a structured, expressive, and unambiguous representation to specify tasks. Specifically, we consider programs, which are composed of control flows (*e.g.* if/else and loops), environmental conditions, as well as corresponding subtasks, as shown in Figure 7.1. Not only do programs have expressiveness by describing diverse situations (*e.g.* a river exists) and the corresponding subtasks which are required to be executed (*e.g.* mining wood), but they are also unambiguous due to their explicit scoping. To study the effectiveness of using programs as task specifications, we introduce a new problem, where we aim to develop a framework which learns to comprehend a task specified by a program as well as perceive and interact with the environment to accomplish the task.

To address this problem, we propose a modular framework, *program guided agent*, which exploits the structural nature of programs to decompose and execute them as well as learn to ground program tokens with the environment. Specifically, our framework consists of three modules: (1) a program interpreter that leverages a grammar provided by the programming language to parse and execute a program, (2) a perception module that learns to respond to conditional queries (*e.g.* `is_there[River]`) produced by the interpreter, and (3) a policy that learns to fulfill a variety of subtasks (*e.g.* `mine(Wood)`) extracted from the program by the interpreter. To effectively instruct the policy with symbolically represented subtasks, we introduce a learned modulation mechanism that leverages a subtask to modulate the encoded state features instead of concatenating them. Our framework (shown in Figure 7.3) utilizes a *rule-based* program interpreter to deal with programs as well as *learning* perception module and policy when it is necessary to perceive or interact with the environment. With this modularity, our framework can generalize to more complex program-specified tasks without additional learning.

To evaluate the proposed framework, we consider a Minecraft-inspired 2D gridworld environment, where an agent can navigate itself across different terrains and interact with objects, similar to [11, 281]. A

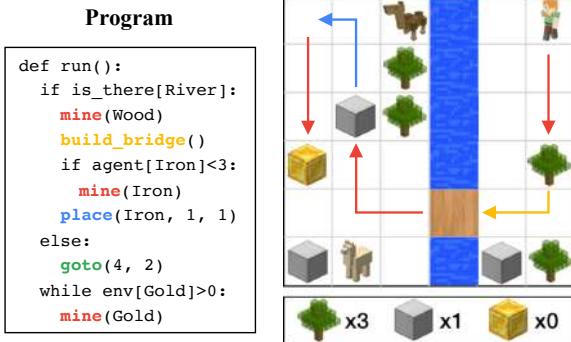


Figure 7.1: An illustration of the proposed problem. We are interested in learning to fulfill tasks specified by written programs. A program consists of control flows (e.g. `if`, `while`), branching conditions (e.g. `is_there[River]`), and subtasks (e.g. `mine(Wood)`).

Program $p := \text{def run}() : s$
Statement $s := \text{while}(c) : (s) \mid b \mid \text{loop}(i) : (s)$
 $\quad \mid \text{if}(c) : (s) \mid \text{elseif}(c) : (s) \mid \text{else} : (s)$
Item $t := \text{Gold} \mid \text{Wood} \mid \text{Iron}$
Terrain $u := \text{Bridge} \mid \text{River} \mid \text{Merchant} \mid \text{Wall} \mid \text{Flat}$
Operators $o := > \geq == < \leq$
Numbers $i := \text{A positive integer or zero}$
Perception $h := \text{agent}[t] \mid \text{env}[t] \mid \text{is_there}[t] \mid \text{is_there}[u]$
Behavior $b := \text{mine}(t) \mid \text{goto}(i, i)$
 $\quad \mid \text{place}(t, i, i) \mid \text{build_bridge}() \mid \text{sell}(t)$
Conditions $c := h[t] o i \mid h[u] o i$

Figure 7.2: The domain-specific language (DSL) for constructing programs. Each program is composed of domain dependent perception, subtasks, and control flows.

corresponding domain-specific language (DSL) defines the rules of constructing programs for instructing an agent to accomplish certain tasks. Our proposed framework demonstrates superior generalization ability – learning from simpler tasks while generalizing to complex tasks. We also conduct extensive analysis on various end-to-end learning models which learns from not only program instructions but also natural language descriptions. Furthermore, our proposed learned policy modulation mechanism yields a better learning efficiency compared to other commonly used methods that simply concatenate a state and goal.

7.2 Related Work

Learning from language instructions. Prior works have investigated leveraging natural languages to specify tasks on a wide range of applications, including navigation [194, 295, 86, 312, 272, 32, 33, 195, 295], spatial reasoning for goal reaching [127], game playing [135, 85, 249], and grounding visual concepts [135, 21, 10]. However, natural language descriptions can often be ambiguous even to humans. Moreover, it is not clear how end-to-end learning agents trained with simpler instructions can generalize well to much more complex ones. In contrast, we propose to utilize a precise and structured representation, programs, to specify tasks.

Learning from demonstrations. When a task cannot be easily described in languages (e.g. object texture or geometry), expert demonstrations offer an alternative way to provide instructions. Prior works have explored learning from video demonstrations [82, 344, 331, 225, 283, 19] or expert trajectories [69, 322]. However, demonstrations can be expensive to obtain and are less expressive about diverging behaviors of a complex task, which are better captured by control flow in programs. Moreover, editing demonstrations such as rearranging the order of subtasks is often difficult.

Program induction and synthesis. To learn acquire programmatic skills such as digit addition and string transformations and achieve better generalization, program induction methods [331, 61, 208, 95, 132, 247, 38, 328] aim to implicitly induce the underlying programs to mimic the behaviors demonstrated in task specifications (e.g. input/output pairs, demonstrations). On the other hand, program synthesis methods [31, 219, 63, 46, 273, 35, 175, 286, 171, 169] explicitly synthesize the underlying programs and execute the programs to perform the tasks. Instead of trying to infer programs from task specifications, we are interested in explicitly executing programs. Also, our framework can potentially be leveraged to obtain program execution results for evaluating program synthesis frameworks when no program executor is available (e.g. programs describe real-world activities instead of behaviors in simulation).

Symbolic planning and programmable agent. Classical symbolic planning concerns the problem of achieving a goal state from an initial state through a series of symbolically represented executions [90, 144]. Our work shares a similar spirit but assume a task (*i.e.* a program) is given, where the agent is required to learn to ground symbolic concepts [185, 107] and follow the control flow. Executing programs with reinforcement learning has been studied in programmable hierarchies of abstract machines [221, 8, 9], which provide partial descriptions and subroutines of the desired task. [59, 155] train agents to execute declarative programs by grounding these well-structured languages in their learning environments. In contrast, our modular framework consists of modules for perceiving the environment and interacting with

it by following an imperative program which specifies the task. An extended discussion on the related work can be found in Section 7.7.3.

7.3 Problem Formulation

We are interested in learning to comprehend and execute an instruction specified by a program to fulfill the desired task. In this section, we formally describe our definition of programs, the family of Markov Decision Processes (MDPs), and the problem formulation.

Program. The programs considered in this work are defined based on a Domain-Specific Language (DSL) as shown in Figure 7.2. The DSL is composed of perception primitives, action primitives, and control flow. A perception primitive indicates circumstances in the environment (e.g. `is_there(River)`, and `agent[Gold]<3`) that can be perceived by an agent, while an action primitive defines a subtask that describes a certain behavior (e.g. `mine(Gold)`, and `goto(1,1)`). Control flow includes `if/else` statements, loops, and Boolean/logical operators to compose more sophisticated conditions. A program p is a deterministic function that outputs a desired behavior (*i.e.* subtask) given a history of states $o_t = p(H_j)$, where $H_j = \{s_1, \dots, s_t\}$ is a state history with $s \in \mathcal{S}$ denoting a state of the environment, and $o \in \mathcal{O}$ denotes an instructed behavior (subtask). We denote a program as $p \sim \mathcal{P}$, an infinite program set containing all executable programs given a DSL. Note that a discussion on the DSL design principle can be found in Section 7.7.2.

MDPs. We consider a family of finite-horizon discounted MDPs in a shared environment, specified by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{T}, \mathcal{R}, \rho, \gamma)$, where \mathcal{S} denotes a set of states, \mathcal{A} denotes a set of low-level actions an agent can take, \mathcal{P} denotes a set of programs specifying instructions, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ denotes a transition probability distribution, \mathcal{R} denotes a task-specific reward function, ρ denotes an initial state distribution, and γ denotes a discount factor. For a fixed sequence $\{(s_0, a_0), \dots, (s_t, a_t)\}$ of states and actions obtained

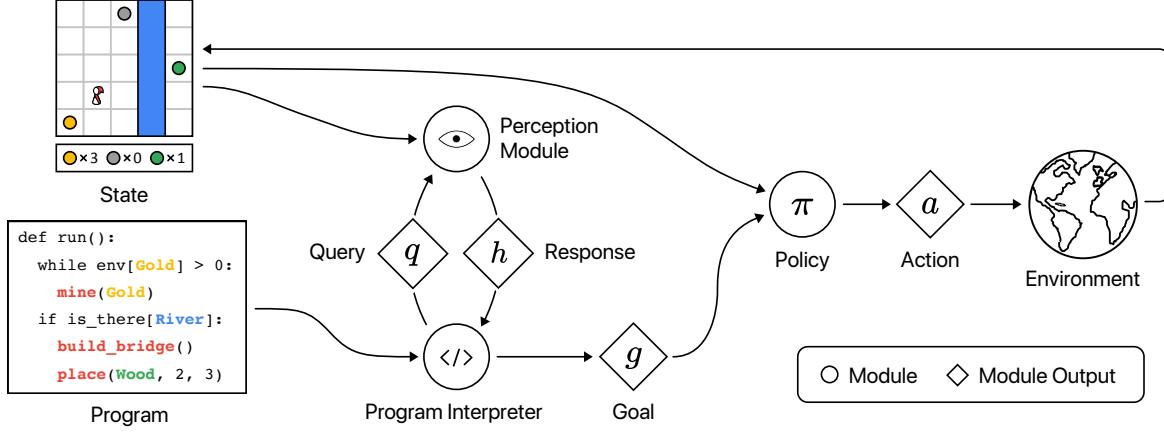


Figure 7.3: Program Guided Agent. The proposed modular framework comprehends and fulfills a desired task specified by a program. The program interpreter executes the program by altering between querying the perception module with a query q when an environment condition is encountered (e.g. $\text{env}[\text{Gold}] > 0$, $\text{is_there}[\text{River}]$) and instructing a policy when it needs to fulfill a goal/subtask g (e.g. $\text{mine}(\text{Gold})$, $\text{build_bridge}()$). The perception module produces a response h to answer the query, determining which paths in the program should be chosen. The policy takes a sequence of low-level actions a (e.g. moveUp , moveLeft , Pickup) interacting with the environment to accomplish the given subtask (e.g. $\text{mine}(\text{Gold})$).

from a rollout of a given policy π , the performance of the policy is evaluated based on a discounted return $\sum_{t=0}^T \gamma^t r_t$, where T is the horizon of the episode.

Problem Formulation. We consider developing a framework which can comprehend and fulfill an instruction specified by a program. Specifically, we consider a sampled MDP with a program describing the desired task. Addressing this task requires the ability to keep track of which parts of the program are finished and which parts are not, perceiving the environment and deciding which paths in the program to take, and performing actions interacting with the environment to fulfill subtasks.

7.4 Approach

Accomplishing an instructed task described by a program requires (1) executing the program control flow and conditions, (2) recognizing the situations to infer which path in the program should be chosen, and (3) executing a series of actions interacting with the environment to fulfill the subtasks. Based on this intuition, we design a modular framework with three modules:

- **Program interpreter** (Section 7.4.1) reads a program and executes it by querying a perception module with environment conditions (e.g. `env[Gold]>0`) and instructing the policy with subtasks (e.g. `mine(Gold)`).
- **Perception module** (Section 7.4.2) responds to perception queries (e.g. `env[Gold]>0`) by examining the observation and predicting responses (e.g. `true`).
- **Policy (action module)** (Section 7.4.3) performs low-level actions (e.g. `moveUp`, `moveLeft`, `pickUp`) to fulfill the symbolically represented subtasks (e.g. `mine(Gold)`) provided by the program interpreter.

Our key insight is to only *learn* a module when its input or output is associated with the environment (*i.e.* a function approximator is needed) – the perception module learns to ground the queries to its observation and answer them; the policy learns to ground the symbolically represented subtasks and interact with the environment in a trial-and-error way (Section 7.4.4). On the other hand, we do not learn the program interpreter; instead, we utilize a *rule-based* parser to execute programs. An overview of the proposed framework is illustrated in Figure 7.3.

7.4.1 Program Interpreter

To execute a program instruction, we group program tokens into three main categories: (1) **subtasks** indicate what the agent should perform (e.g. `mine(Gold)`), (2) **perceptions** the essential information extracted from the environment (e.g. `env[Gold]>0`), and (3) **control flows** determine which paths in a program should be taken according to the perceived information (*i.e.* perceptions). Then, we devise a *program interpreter*, which can execute and keep track of the progress by leveraging the structure of programs. Specifically, it consists of a program line parser and a program executor. The parser first transforms the program into a program tree by representing each line of a program as a tree node. Each node is either a leaf node (subtask) or a non-leaf node (perception or control flow) that has various subroutines as children nodes. The executor then performs a pre-order traversal on the program tree to execute the program, utilizing the

parsed contents to alternate between querying the perception module when an environment condition is encountered and instructing the policy when it reaches to a leaf node (subtask). The details and the algorithm are summarized in Section 7.7.1. Note that the *program interpreter* is a rule-based algorithm instead of a learning module.

7.4.2 Perception Module

Determining which paths should be chosen when executing a program requires grounding a symbolically represented query (e.g. `is_there[River]` can be represented as a sequence of symbols) and perceiving the environment. To this end, we employ a *perception module* Φ that learns to map a query and current observation to a response: $h = \Phi(q, s)$, where q denotes a query, and h denotes the corresponding perception output (e.g. `true/false`). Note that we focus on Boolean perception outputs in this paper, but a more generic perception type can be used (e.g. object attributes such as color, shape, and size).

7.4.3 Policy

When program execution reaches a subtask/leaf node (e.g. `mine(Gold)`), the agent is required to take a sequence of low-level actions (e.g. `moveUp`, `moveLeft`, `Pickup`) to interact with the environment to fulfill it. To enable the execution, we employ a multitask policy π (*i.e.* action module) which is instructed by a symbolic goal (e.g. `mine(Gold)`) provided by the program interpreter indicating the details of the corresponding subtask. To learn to perform different subtasks, we train the policy using actor-critic reinforcement learning, which takes a goal vector g and an environment state s and outputs a probabilistic distribution a for low-level actions $a \sim \pi(s_t, g_t | \theta)$. The value estimator used for our policy optimization is also goal-conditioned: $V_\pi(s_t, g_t) = \mathbb{E}[\sum_t \gamma^t R_t | s_0 = s, \pi, g_t]$.

While the most common way to feed a state and goal to a policy parameterized by a neural network is to concatenate them in a raw space or a latent space, we find this less effective when the policy has to

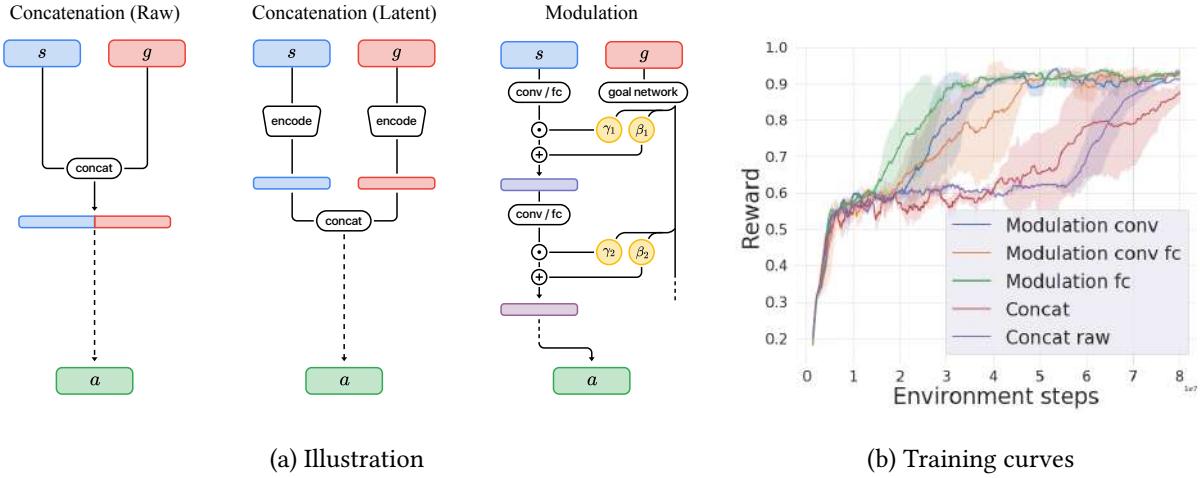


Figure 7.4: Learning a multitask policy via learned modulation. (a) A multitask policy takes both a state s and a goal specification g as inputs and produces an action distribution $a \sim \pi(s, g)$. Instead of simply concatenating the state and goal in a raw space or a latent space, we propose to modulate state features e_s using the goal. Specifically, the goal network learns to predict affine transform parameters γ and β to modulate the state features $\hat{e}_s = \gamma \cdot e_s + \beta$. Then, the final layers use the modulated features to predict actions. (b) We experiment different ways of feeding a state and goal for learning a multitask policy. The training curves demonstrate that all modulation variants, including modulating state feature maps of convolutional layers (*Modulation conv*), modulating state feature vectors of fully-connected layers (*Modulation fc*), or both (*Modulation conv fc*), are more efficient than concatenating the state and the goal in a raw space (*Concat raw*) or a latent space (*Concat*).

learn a diverse set of tasks. Therefore, we propose a modulation mechanism to effectively learn the policy.

Specifically, we employ a goal network to encode the goal and compute affine transform parameters γ and β , which are used to modulate state features e_s to $\hat{e}_s = \gamma \cdot e_s + \beta$. Then, the modulated features \hat{e}_s are used to predict action a and value V . With the modulation mechanism, the goal network learns to activate state features related to the current goal and deactivate others. An illustration is shown in Figure 7.4 (a). A more detailed discussion of the related works that utilize similar learned modulation mechanisms can be found in Section 7.7.4.

7.4.4 Learning

To follow a program by perceiving the environment and taking actions to interact with it, we employ two learning modules: a perception module and a policy. In this section, we discuss how each module is trained,

their training objectives, and optimization methods. More training details and the architectures can be found in Section 7.7.5.4.

7.4.4.1 Perception Module

We formulate training the perception module as a supervised learning task. Given tuples of (query q , state s , ground truth perception h_{gt}), we train a neural network Φ to predict the perception output h by optimizing the binary cross-entropy loss: $\mathcal{L}_{CE} = -h_{gt}\log(h) - (1 - h_{gt})\log(1 - h)$. A query such as `is_there[River]` is represented as a sequence of symbols. Note that when perception describes more than a Boolean, training the perception module can be done by optimizing other losses such as categorical cross-entropy loss. We train the perception module only on the queries appearing in the training programs with randomly sampled states, requiring it to generalize to novel queries to perform well in executing testing programs.

7.4.4.2 Policy

We train the policy using Advantage Actor-Critic (A2C) [198, 64], which is commonly used for gridworld environments with discrete action spaces. A2C computes policy gradients $A_t \nabla_\theta \log \pi_\theta(a_t|s_t, g_t)$, where $A_t = R_t - V(s_t, g_t)$ is the advantage function based on empirical return R_t starting from s_t and learned value estimator $V(s_t, g_t)$ conditioning on the goal vector g_t . We denote the learning rate as α , and the policy update rule is as follows:

$$\theta \leftarrow \theta + \alpha (A_t \nabla_\theta \log \pi_\theta(a_t|s_t, g_t) + \beta \nabla_\theta H_{\pi_\theta}), \quad (7.1)$$

where H_{π_θ} denotes the policy entropy, where maximizing it improves overall exploration, and β determines the strength of the entropy regularization term.

7.5 Experiments

Our experiments aim to answer the following questions: (1) Can our proposed framework learn to perform tasks specified by programs? (2) Can our modular framework generalize better to more complex tasks compared to end-to-end learning models? (3) How well can a variety of end-to-end learning models (*e.g.* LSTM, Tree-RNN, Transformer) learn from programs and natural language instructions? (4) Is the proposed learned modulation more efficient to learn a multitask (multi-goal) policy than simply concatenating a state and goal?

7.5.1 Experimental Setups

7.5.1.1 Environment

To evaluate the proposed framework in an environment where an agent can perceive diverse scenarios and interact with the environment to perform various subtasks, we construct a discrete Minecraft-inspired gridworld environment, similar to [11, 281]. As illustrated in Figure 7.1, the agent can navigate through a grid world and interact with resources (*e.g.* Wood, Iron, Gold) and obstacles (*e.g.* River, Wall), build tools (*e.g.* Bridge), and sell resources to a merchant visualized as an alpaca. The environment gives a sparse task completion reward of +1 when an instruction (*i.e.* an entire program or natural language instruction) is successfully executed. More details can be found in Section 7.7.5.1.

7.5.1.2 Task Instructions

Programs. We sample 4,500 programs using our DSL and split them into 4,000 training programs (*train*) and 500 testing programs (*test*). To examine the framework’s ability to generalize to more complex instructions, we generate 500 programs which are twice longer and contains more condition branches on average to construct a harder testing set (*test-complex*).

Natural language instructions. To obtain the natural language counterparts of those instructions, we asked annotators to construct natural language translations of all the programs. The data collection details, as well as sample programs and their corresponding natural language translations, can be found in Section 7.7.5.3, and Figure 7.10 respectively. We include a brief discussion on how annotated natural language instructions can be ambiguously interpreted as several valid programs.

7.5.2 Training

During training, we randomly sample programs from the training set as well as randomly sample an environment state to execute the program interpreter. The program interpreter produces a goal to instruct the policy when encountering a subtask in the program. The policy takes actions $a \sim \pi(s, g)$ and receive reward +1 only when the entire program is completed. While we do not explicitly introduce a curriculum like [11], this setup naturally induces a curriculum where the policy first learns to solve simpler programs and gain a better understanding of subtasks by obtaining the task completion, which eventually allows the policy to complete more complex programs. Note that the perception module is pre-trained beforehand in a supervised manner. More training details can be found in Section 7.7.5.7.

7.5.3 End-to-end Learning Models

In contrast to the proposed modular framework, we experiment with a variety of end-to-end learning models. Considering programs and natural language instructions as sequences of tokens, we investigate two types of state-of-the-art sequence encoders: LSTM [115] (*Sq-LSTM*), and Transformers [305, 62] (*Transformer*). To leverage the explicit structure of programs, we also investigate encoding programs using a generalization of RNNs for tree-structured input [293, 7] (*Tree-RNN*). All the models are trained using A2C. The details of these architectures can be found in Section 7.7.5.4.

Table 7.1: Task completion rate. For each method, we iterate over all the programs in a testing set by randomly sampling ten initial environment states and running three models trained using different random seeds for this method. The averaged task completion rates and their standard deviations are reported. Note that all the end-to-end learning models learning from natural language descriptions and programs suffer from a significant performance drop when evaluated on the more complex testing set.

Instruction		Natural language descriptions		Programs				
	Method	Seq-LSTM	Transformer	Seq-LSTM	Tree-RNN	Transformer	Ours (concat)	Ours
Dataset	test	54.9±1.8%	52.5±2.6%	56.7±1.9%	50.1±1.2%	49.4±1.6%	88.6±0.8%	94.0±0.5%
	test-complex	32.4±4.9%	38.2±2.6%	38.8±1.2%	42.2±2.4%	40.9±1.5%	85.2±0.8%	91.8±0.2%
Generalization gap		40.9%	27.2%	31.6%	15.8%	17.2%	3.8%	2.3%

7.5.4 Results

7.5.4.1 Task Completion

We train the proposed framework and the end-to-end learning models on training programs and evaluate their performances using the percentage of completed instructions on *test* and *test-complex* sets (shown in Table 7.1). Our proposed framework achieves a satisfactory *test* performance and only suffers a negligible drop (*i.e.* generalization gap) when it is evaluated on *test-complex* set. This can be attributed to the modular design, which explicitly utilizes the structure and grammar of programs, allowing the two learning modules (*i.e.* perception and policy) to focus on their local jobs. A more detailed failure analysis can be found in Section 7.7.5.6.

On the other hand, all the end-to-end learning models suffer a significant performance drop between *test* and *test-complex* sets, while it is less significant for the models learning from programs, potentially indicating that models learning from instructions with explicit structures can generalize to complex instructions better. Among them, Seq-LSTM achieves the best results on *test* set, but performs the worst on the *test-complex* set. Transformer has smaller generalization gaps, which could be attributed to their multi-head attention mechanism, capturing the instruction semantics better. By leveraging the explicit structure of programs, Tree-RNN achieves the best generalization performance.

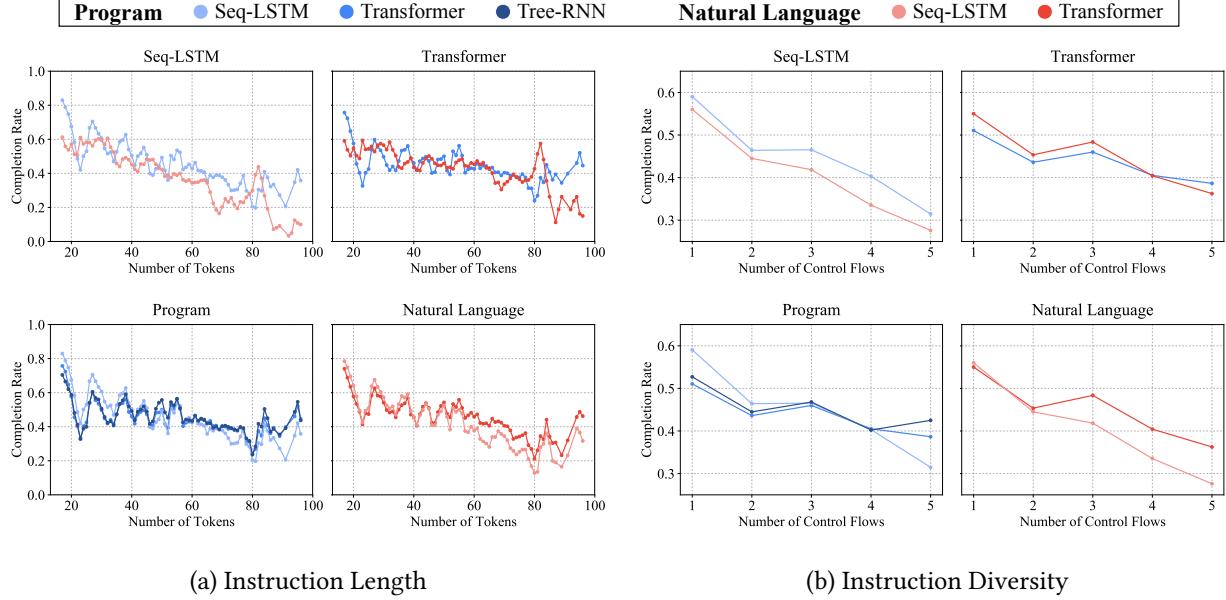


Figure 7.5: Analysis on end-to-end learning models: (a) Models learning from programs generalize better to longer instructions. Transformer is more robust to longer instructions (Upper). Tree-RNN exploiting the program structure generalizes the best, but performs worst for shorter programs (Lower). (b) Seq-LSTM learning from both instructions performs worse as the diversity increases. Transformer learns better from natural language when the instructions are less diverse (Upper). Transformer and Tree-RNN learning from programs are more consistent as the diversity increases, yet perform worse on less diverse instructions (Lower).

7.5.4.2 Analysis

An analysis on the end-to-end learning models with respect to varying instruction length and complexity is shown in Figure 7.5, where all the instructions from *test* and *test-complex* sets are considered.

Instruction length. As shown in Figure 7.5 (a), both Seq-LSTM and Transformer suffer from a performance drop as the instruction length increases. Seq-LSTM performs better when instructions are shorter, but suffers from generalizing to longer instructions. On the other hand, Transformer may learn on a more semantic level, which leads to similar overall performances across two types of instructions. Tree-RNN leverages the structure of programs and achieves a better performance.

Instruction diversity. We define the diversity of a program based on the number of control flows it contains (*i.e.* number of branches). Figure 7.5 (b) shows a clear trend of performance drop of all the models. Transformer is more robust to diverse instructions which could be attributed to its better ability to learn

the semantics. While Seq-LSTM learning from programs are consistently better across different levels of diversities, Tree-RNN demonstrates the most consistent performances.

7.5.5 Policy Modulation

We investigate if learning a multitask policy with the learned modulation mechanism is more effective. We compare against the two most commonly used methods: concatenating a state and goal in a raw space (*Concat raw*) or a latent space (*Concat*). An illustration is shown in Figure 7.4 (a). Since our state contains an environment map, which is encoded by a CNN and MLP, we experiment modulating convoluted feature maps (*Modulation conv*) or feature vectors (*Modulation fc*) or both (*Modulation conv fc*). Figure 7.4 (b) demonstrate that the proposed policy modulation mechanism is more sample efficient. Table 7.1 shows that the multitask policy learning using modulation achieves better performance on task completion.

7.6 Conclusion

We propose to utilize programs, structured in a formal language, as an expressive and precise way to specify tasks instead of commonly used natural language instructions. We introduce the problem of developing a framework that can comprehend a program as well as perceive and interact with the environment to accomplish the desired task. To address this problem, we devise a modular framework, *program guided agent*, which executes programs with a program interpreter by altering between querying a perception module when a branching condition is encountered and instructing a policy to fulfill subtasks. We employ a policy modulation mechanism to improve the efficiency of learning the multitask policy. The experimental results on a 2D Minecraft environment demonstrate that the proposed framework learns to reliably fulfill program instructions and generalize well to more complex instructions without additional training. We also investigate the performance of various models that learn from programs and natural language descriptions in an end-to-end fashion.

7.7 Appendix

7.7.1 Program Execution

We describe our program interpreter in Section 7.4 and provide more details in this section. The program instruction considered in this work contains the following three components: (1) subtasks, (2) perceptions, and (3) control flows. Accordingly, our *Program Interpreter* is designed to consist of (1) a parser to parse each line of the program following the grammar defined by our DSL in Figure 7.2, and (2) a program executor which executes the program conditioning on the parsed contents. The interpreter transforms the program into a tree-like object by exploiting its structure (*i.e.* scopes) and then utilizes the parsed contents to traverse it to execute the program.

A program tree is built by representing each line of a program as a tree node. Each tree node is a data structure containing members: (1) **node.line**, the original line in the program, (2) **node.isLeaf()**, if the current node is a leaf node (*i.e.* subtask), and (3) **node.children**, all the subroutines of the current node (*i.e.* the processes under the scope of the current line of program). The interpreter will parse according to the original line contained in the node, and decide whether to call the policy if it is a leaf node (subtask) or produce a query to call the perception module, deciding which child node (subroutine) to go into.

The subroutines of a node should correspond to proper scoping of the program. For example, in Figure 1 in the main paper, the line `if is_there[River]` has subroutines `mine(Wood)`, `build_bridge()`, if `agent[Iron]<3`, and `place(Iron,1,1)`, but not `mine(Iron)`, which should be `if agent[Iron]<3`'s subroutine.

Once the program tree is built, the program executor will perform a pre-order traversal to initiate the execution. Algorithm 3 summarizes the details of the program execution utilizing the transformed program tree.

Algorithm 3 Program Execution

Require: P : program to be executed

Require: s : environmental state

Require: π : agent policy parameterized by θ , Φ : perception module

Require: node: has member node.line as the original program line and children nodes node.children

```
1: procedure EXECUTE(node)
2:   if node.isLeaf() then
3:     subtask = parse_subtask(node.line)
4:      $\pi(\text{subtask}, \theta)$                                  $\triangleright$  Calls the agent policy to execute the subtask
5:   else
6:     control_flow, perception_query = parse_ctrl_percept(node.line)
7:      $h = \Phi(\text{perception\_query}, s)$                  $\triangleright$  Calls the perception module with a query and state
8:     control_flow  $h$                                    $\triangleright$  e.g. if, while, loop, calls the subroutines depending on  $h$ 
9:     for child in node.children do
10:      EXECUTE(child)
11:    end for
12:  end if
13: end procedure
```

7.7.2 DSL Design Principle

Since different domains require different DSLs, we aim to design our DSL by following a design principle that would potentially allow us to easily adapt our DSL to different domain. Specifically, we develop a DSL design principle that considers a general setting where an agent can perceive and interact with the environment to fulfill some tasks. Accordingly, our DSL consist of control flows, perceptions, and actions. While control flows are domain independent, perceptions and actions can be designed based on the domain of interest, which would require certain expertise and domain knowledge. We aim to design our DSL that

is (1) intuitive: the actions and perceptions are intuitively align with human common sense, (2) modular: actions are reasonably distinct and can be used to compose more complex behaviors, and (3) hierarchical: a proper level of abstraction that enables describing long-horizon tasks.

7.7.3 Extended Related Work

We present an extended discussion of the related work in this section.

Multitask reinforcement learning. To achieve multi-task reinforcement learning, previous works devised hierarchical approaches where an RL agent is trained to achieve a series of subtasks to accomplish the task. In [11], a sequence of policy sketches is predefined to guide an agent towards the desired goal by leveraging modularized neural network policies. [216] propose to learn a controller to predict to either proceed, revert, or stay at a current subgoal, which is sampled from a list of simple symbolic instructions. In this paper, hierarchical tasks are described by programs with increased diversity through branching conditions, and therefore our framework is required to determine which branches in a program should be executed. On the other hand, the framework proposed by [281] requires a subtask graph describing a set of subtasks and their dependencies and aims to find the optimal subtask to execute. This is different from our problem formulation where the agent is asked to follow a given program/procedure.

Hierarchical reinforcement learning. Our work is also closely related to hierarchical reinforcement learning, where a meta-controller learns to predict which sub-policy to take at each time step [148, 20, 66, 84, 308, 160, 23, 203, 184]. Previous works also investigated in explicitly specifying sub-policy with symbolic representations for meta-controller to utilize, or an explicit selection process of lower-level motor skills [201, 298].

Programmable agents. We would like to emphasize that our work differs from programmable agents [59] in motivation, problem formulations, proposed methods, and contributions. First, [59] concern declarative programs which specify what to be computed (e.g. a target object in a reaching task). However,

the programs considered in our work are imperative, which how this is to be computed (i.e. a procedure). Also, [59] consider only one-liner programs that contain only AND, OR, and object attributes. On the other hand, we consider programs that are much longer and describe more complex procedures. While [59] aim to generalize to novel combinations of object attributes, our work is mainly interested in generalizing to more complex tasks (i.e. programs) by leveraging the structure of programs.

Programs vs. natural language instructions. In this work, we advocate utilizing programs as a task representation and propose a modular framework that can leverage the structure of programs to address this problem. Yet, natural language instructions enjoy better accessibility and are more intuitive to users who do not have experience in programming languages. While addressing the accessibility of programs or converting a natural language instruction to a more structural form is beyond the scope of this work, we look forward to future research that leverages the strengths of both programs and natural language instructions by bridging the gap between these two representations, such as synthesizing programs from natural language [171, 60, 245], semantic parsing that bridges unstructured languages and structural formal languages [343, 342], and naturalizing program [320].

7.7.4 Discussions on Learned Modulation Mechanisms

To fuse the information from an input domain (e.g. an image) with another condition domain (e.g. a language query, image such as segmentation map, noise, etc.), a wide range of works have demonstrated the effectiveness of predicting affine transforms based on the condition to scale and bias the input in visual question answering [231, 230], image synthesis [6, 136, 220, 121], style transfer [71], recognition [119, 329], reading comprehension [65], few-shot learning [217, 158], etc. Many of those works present an extensive ablation study to compare the learned modulation against traditional ways to merge the information from the input and condition domains.

Recently, a few works have employed a similar learned modulation technique to reinforcement learning frameworks on learning to follow language instruction [21] and meta-reinforcement learning [315, 314]. However, there has not been a comprehensive ablation study that suggests fusing the information from the input domain (*e.g.* a state) and the condition domain (*e.g.* a goal or a task embedding) for the reinforcement learning setting. In this work, we conduct an ablation study in our 2D Minecraft environment where an agent is required to fulfill a navigational task specified by a program and show the effectiveness of learning to modulate input features with symbolically represented goal as well as present a number of modulation variations (*i.e.* modulating the fully-connected layers or the convolutional layers or both). We look forward to future research that verifies if this learned modulation mechanism is effective in dealing with more complex domains such as robot manipulation or locomotion.

7.7.5 Additional Experimental Details

7.7.5.1 Environment Details

In the following paragraphs, we provide some details of the environment used in this work.

Objects in the environment. The major environmental resources that the agent can interact with are: wood, gold, and iron. There is a certain probability that the environment will contain a `river`, which the agent cannot go across unless a `bridge` is built (or pre-built). The environment is surrounded by brick walls, which draws the boundaries of the world.

Agent action space. The agent’s actions are (1) **crafting actions**: including mining (collecting resources), placing, building a bridge, and selling an item; and (2) **motor actions**: including moving to four directions (up, down, left, right). The crafting actions are only allowed on the current grid cell the agent is standing on, *e.g.* the subtask `mine(gold)` requires the agent to navigate to a specific location containing a `gold` with motor actions, and then perform the crafting action `mine` at the current location. To build a

bridge, the agent should consume one of the wood it possesses. To sell an item, the agent needs to travel to a merchant. With certain probabilities, there can be 2 to 4 merchants at different locations.

Initialization. During training, when each training program is sampled, a valid environment will be randomly initialized, where validity refers to the property that the agent will be able to successfully follow the program with sufficiently provided environmental resources. At test time, we pre-sample 20 valid initialization of the environment with 20 different random seeds to ensure the validity of the two test sets.

Agent observation space (state representations). The state used in our reinforcement learning policy consists of an environment map s_{map} and an inventory status of the agent s_{inv} . s_{map} is of size $10 \times 10 \times 9$, where each channel-wise slice of size $10 \times 10 \times 1$ represents the binary existence of certain objects at a specific location, e.g. if $(3, 4, 2)$ is 1, it means there is a gold at location $(3, 4)$ (environment objects in channel dimension are zero-indexed). The objects represented by the channels are ordered as follows: wood, iron, gold, agent, wall, goal (2-D representation of intended goal coordinates), river, bridge, and merchant. The agent inventory status s_{inv} is augmented with the agent location, resulting in a 1-d integer vector of size 5. The ordered entry of such vector is as follows: agent's wood counts, agent's iron counts, agent's gold counts, agent's location coordinate x, and then y.

Goal representations. For our proposed framework, we represent the goal of the subtask as an 1-D vector of size 10, produced by the program interpreter. The first five entries of the goal vector is a one-hot representation of the subtask: goto, place, mine, sell, build_bridge. The 6th to 8th entries are one-hot representation of the resources: wood, iron, and gold. The last two entries are the intended goal locations. For example, `place(iron, 3, 5)` will be represented as $[0, 1, 0, 0, 0, 0, 1, 0, 3, 5]$. For end-to-end learning models, such goal representation is produced by the input encoder as a continuous latent vector representation.

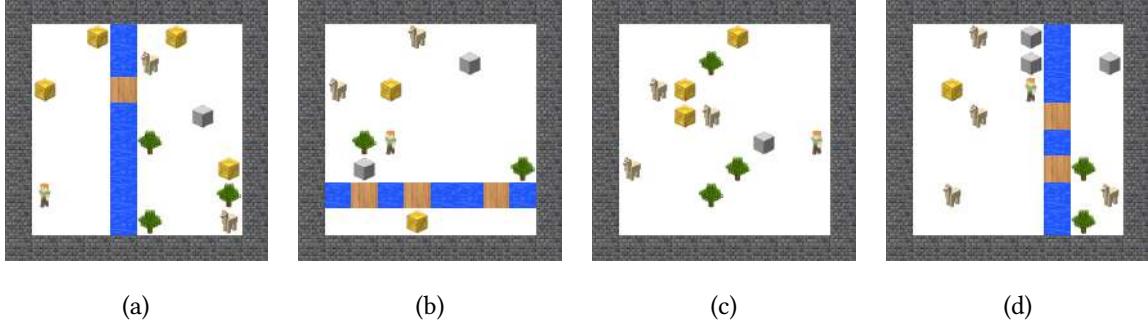


Figure 7.6: Exemplar rendered environment map. The agent, objects, and stuff are represented as blocks with their corresponding textures. Specifically, the agent is represented as a female character. gold is represented as a golden block, wood is shown as a tree, and iron is represented as a silver block. River is shown as a blue grid with water texture while bridge is presented as wooden grid. merchant is shown as an alpaca, which is supposed to transport the sold objects. Notice that there are 2 merchants in (a) and (b), while (c) and (d) contains 3 and 4 of them, respectively. The boundaries of the map are shown as brick walls.

Exemplar environment maps. We show several exemplary rendered environment maps in Figure 7.6.

As can be seen, the essential resources such as wood, gold, iron are represented as block objects, where the merchant is depicted by an alpaca. The agent is shown as a female human character. River grids, with bridge blocks built on it is shown as the blue grid cells, where the bridge which should be transformed by wood is of wooden texture. The Boundaries can be seen in the surroundings represented as brick wall grids.

7.7.5.2 Ground Truth Perceptions for End-to-end Learning Baselines

Since we train our perception module using ground truth information, we provide the ground truth perception information to all our baselines yet failed to elaborate this in the original paper. Specifically, at every time step, we feed the ground truth perception (*i.e.* the answer to the queries such as `env[Gold]>0` and `is_there[River]`) to the baseline models. The ground truth perception is represented as a vector that has a dimension of the number of all possible queries, and each element corresponds to a binary answer to a query. Therefore, the baseline models can learn to utilize this ground truth information to infer the desired subtasks. During testing time, the baseline models can still access to all this ground truth perception information, even though it is usually not possible in practice. On the contrary, during testing time, our

perception module predicts the answer to given queries and the performance of the whole framework depends on the predicted answers.

7.7.5.3 Task Instructions Details

Programs. We generate the program sets by sampling program tokens with normal distributions, and constructing them according to the DSL grammar we define. The training set is composed of on average 32 tokens and 4.6 lines; the more complex test set, *i.e. test-complex*, contains on average 65 tokens and 9.8 lines. We include the plotted statistics of various essential properties for the three datasets in Figure 7.11, Figure 7.12, and Figure 7.13, respectively. Note that the maximum indent of a program is the maximum depth of its scope or the height of its transformed program tree. The number of recurring procedures includes both `while` and `loop`.

Natural language instructions. For each of the three program sets, we chunked them into several subsets of programs and assign them to annotators for their corresponding natural language translations. The annotators were instructed to read the provided DSL to understand the details of program syntax as well as some exemplary translations before they are allowed to start the task. The annotators were encouraged to give diverse and colloquial translations to avoid constantly giving dull line-by-line translations. The collected (translated) natural language instructions were then cleansed with spell checks and grammatical errors fixes. On average, the annotators used 27, 28, and 61 words to describe the instructions for the train, test, and test-complex sets respectively. The total vocabulary size of the natural language instructions is of 448.

Qualitative results on natural language analysis. We show several example data points from our testing sets in Figure 7.10. The leftmost column displays natural language instructions, the middle column shows our sampled ground truth programs, while the rightmost column illustrates how language can be ambiguous and lead to possible alternative interpreted programs.

7.7.5.4 Network Architectures

The proposed framework and the end-to-end learning baselines are implemented in TensorFlow [1].

Our framework Perception module. The perception module takes a query q and a state s as input and outputs a response h . A query has a size of 6×186 , since the longest query has a length of 6 and 186 is the dimension of one-hot program tokens. Shorter queries are zero-padded to this size.

The state map s_{map} is encoded by a CNN with four layers with channel size of 32, 64, 96, and 128. Each convolutional layer has kernel size 3 and stride 2 and is followed by ReLU nonlinearity. The final feature map is flattened to a feature vector, denoted as f_m .

The state inventory s_{inv} is encoded by a two-layer MLP with a channel size of 32 for both layers. Each fully-connected layer is followed by ReLU nonlinearity. The resulting feature vector is denoted as f_i .

Each token in the query is first encoded by a two-layer MLP with a channel size of 32 for both layers. Each fully-connected layer is followed by ReLU nonlinearity. Then, all the query token features are concatenated along the feature dimension to a single vector. This vector is then encoded by another two-layer MLP with a channel size of 32 for both layers. Each fully-connected layer is followed by ReLU nonlinearity. The resulting feature vector is denoted as f_q .

All encoded features (f_m , f_i , and f_q) are then concatenated along the feature dimensions to a single vector. This vector is processed by a three-layer MLP with a channel size of 128, 64, and 32. Each fully-connected layer is followed by ReLU nonlinearity. Finally, a linear fully-connected layer produces an output with a size of 1, which should have a higher value if the response of the query is true and lower otherwise.

Policy. The policy takes a goal g and a state s as input and outputs an action distribution a , where the state is encoded by two types of modules: (1) a four-layer CNN encoder to encode the state map s_{map} , and (2) a two-layer MLP to encode the agent inventory status s_{inv} .

The goal g is encoded by a two-layer MLP with a channel size of 64 for both layers. Each fully-connected layer is followed by ReLU nonlinearity. The resulting feature vector is denoted as f_g . Given the encoded goal vector, we employ four linear fully-connected layers to predict modulation parameters $\{\gamma_i, \beta_i\}_{\{1, \dots, 4\}}$ for the state CNN encoder, where γ_1 and β_1 have size 32, γ_2 and β_2 have size 64, γ_3 and β_3 have size 96, and γ_4 and β_4 have size 128. Note that these modulation parameters are predicted for modulating convolutional features (*i.e. modulation conv*). For *modulation fc*, a linear fully-connected layer is used to produce γ^{fc} and β^{fc} with size 64.

A state map s_{map} is encoded by four-layer CNN with channel size of 32, 64, 96, and 128. Each convolutional layer has kernel size 3, strides 2, and is followed by ReLU nonlinearity. After each convolutional layer, the produced feature maps e are modulated to $\gamma \cdot e + \beta$, where γ and β are broadcast along spatial dimensions. The final feature map is flattened to a feature vector and denoted as f_m^π .

A state inventory s_{inv} is encoded by a two-layer MLP with channel size of 64 for both layers. Each fully-connected layer is followed by ReLU nonlinearity. The resulting feature vector is denoted as f_i^π .

The two encoded features (f_m^π and f_i^π) are then concatenated along the feature dimension. Two fully-connected layers are used to process the feature with a channel size of 64 for both layers. Each layer is followed by ReLU nonlinearity. The final encoded feature u is then modulated to $\gamma^{fc} \cdot u + \beta^{fc}$ if *modulation fc* is used.

Finally, the modulated features \hat{u} are used to produce an action distribution a and a predicted value V using two separated MLPs. Each MLP has two fully-connected layers with a channel size of 64 for both layers. A linear layer then outputs a vector with a size of 8 (the number of low-level actions). Another linear layer outputs a vector with a size of 1 as the predicted value.

End-to-end learning models In addition to the input encoder, the end-to-end learning models can utilize a mechanism to remember what subtasks from the instructions have been accomplished. The agent can then explicitly memorize where it stands in the instruction while completing the task. We augment such

memorization mechanism utilizing the memory of another LSTM network, taking as inputs the encoded states throughout the execution trajectory. After agent taking each action, the last hidden state encoding the trajectory up to the current step is used to compute attention scores to pool the outputs of the input encoders. For Tree-RNN encoder, we simply concatenate the hidden representation from memorization LSTM with the root representation of Tree-RNN before feeding them to subsequent layers. The agent policy network then learns to perform task conditioning on this attention-pooled latent instruction vector.

We provide details of our various end-to-end learning models in Table 7.2. Program token embedding is jointly trained with learning the whole module, while GloVe [229] (50-D version) is used for word embedding when instructions are natural languages.

Model	Parameters	Details
Seq-LSTM	0.62M	LSTM size of 128, both program and word embeddings are of dimension 50. Attention LSTM size of 128. Attention weights of size $[256 \times 128]$, with bias of size $[128]$. Word embeddings utilize pre-trained GloVe.
Tree-RNN	0.51M	Program embeddings are of dimension 128. Attention LSTM size of 128. Composition module (to aggregate all the children representation of a node) is of size $[128 \times 128]$, and output projection weights of size $[128 \times 128]$, with bias of size $[128]$. The program embeddings are average pooled across the same program line, so that each line will be mapped to a fixed dimension representation. The composition layer is applied when combining pooled embedding from all the children of a node.
Transformer	2.63M	Number of hidden layers: 2, with 8 attention heads, and intermediate size of 256. Hidden size is 128. No dropout is applied.

Table 7.2: Architectural details for end-to-end learning models

7.7.5.5 Raw RGB Input

To verify if our framework can be extended to using high-dimensional raw state inputs (*i.e.* RGB image) as inputs where a hand-crafted policy or perception module might not be easy to obtain, we performed an

additional experiment where the perception module and the policy are trained on raw RGB inputs instead of the symbolic state representation. The results suggest that our framework can utilize RGB inputs while maintaining similar performance (93.2% on the test set) and generalization ability (91.4% on the test-complex set).

7.7.5.6 Failure Analysis

To gain a better understanding of how our proposed framework and the end-to-end learning models work or fail, we conduct detailed failure analysis on the execution traces of our model. The analysis is organized as follows:

- We first present an analysis of our framework on the subtasks that appear to be the first failed subtask, which immediately leads to failing the whole task. This analysis sheds some light on which subtasks most commonly cause the failure of task execution. (Section 7.7.5.6)
- We show an analysis of how many time steps each successfully executed subtask takes on average for our framework, through which we explain which subtasks we find to be harder than others. (Section 7.7.5.6)
- We show additional visualizations on the completion rates of different end-to-end learning models plotted with metrics not shown in the main paper, where we aim to deliver a more complete view of how these models perform. (Section 7.7.5.6)

First failure rate of subtasks As the first step of failure analysis, we want to get an idea of which subtasks cause the failure of the model in executions more often. To make this possible, we define “first failed subtask” as the first subtask that ends as a failure in an unsuccessful execution of a program. Based on this definition, we further define “first failure rate” as the percentage that an occurrence of a specific subtask turns out to be the first failed subtask of the execution that includes it.

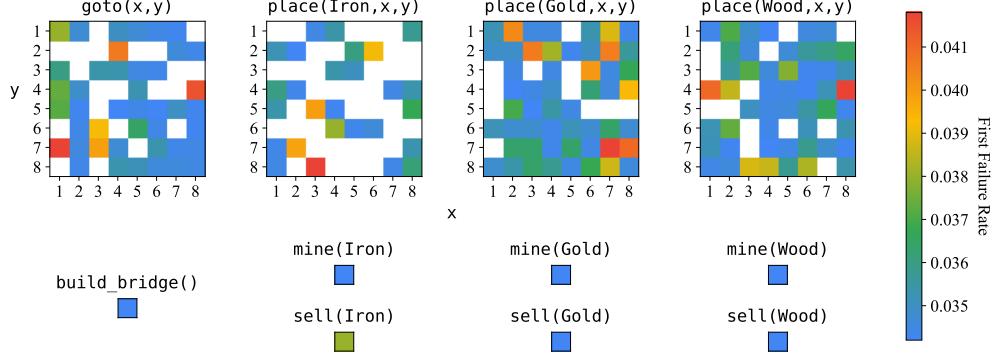


Figure 7.7: First failure rate of subtasks. Every colored grid shows the first failure rate of each subtask. From top-left to bottom-right, each block of grids show the results for subtask category `goto`, `place`, `build_bridge`, `mine`, and `sell`. Warmer colors indicate higher first failure rate; while colder colors indicate lower first failure rate. White grids indicate subtasks that either never occurs as first failed task in any execution or do not exist in the executions that lead to this figure.

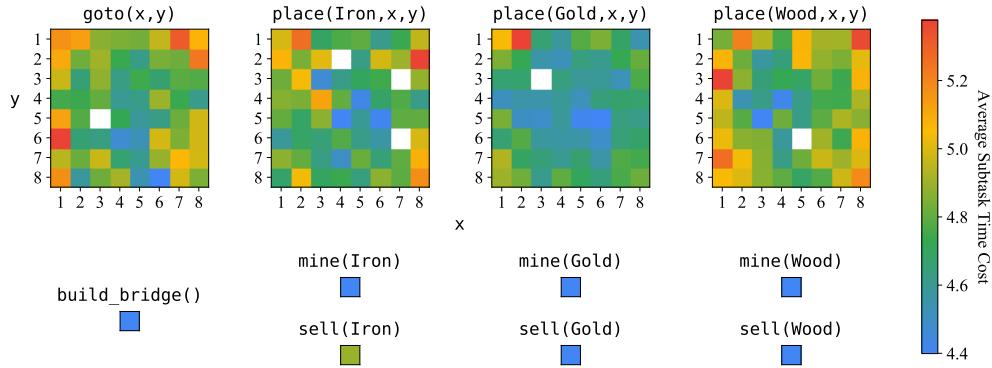


Figure 7.8: Average time cost of subtasks. The setup of this plot is similar to that of Figure 7.7. Warmer colors indicate higher average subtask time cost; while colder colors indicate lower average subtask time cost. White grids indicate subtasks that do not exist in the executions that lead to this figure.

We collect the first failure rate of all subtasks for the result we obtain from running our full model over the more complex test set, *i.e.* `test-complex`. The results are plotted in a visually interpretable format in Figure 7.7. As seen in the figure, subtasks in `goto` and `place` categories are more likely to be the first failed subtask than subtasks in `build_bridge`, `mine`, and `sell` categories. Within the `goto` and `place` subtask categories, subtasks requiring the agent to navigate to grid cells nearby the border of the world has a higher first failure rate than ones near the center of the world. This shows that these tasks mentioned above are more prone to failure than other subtasks.

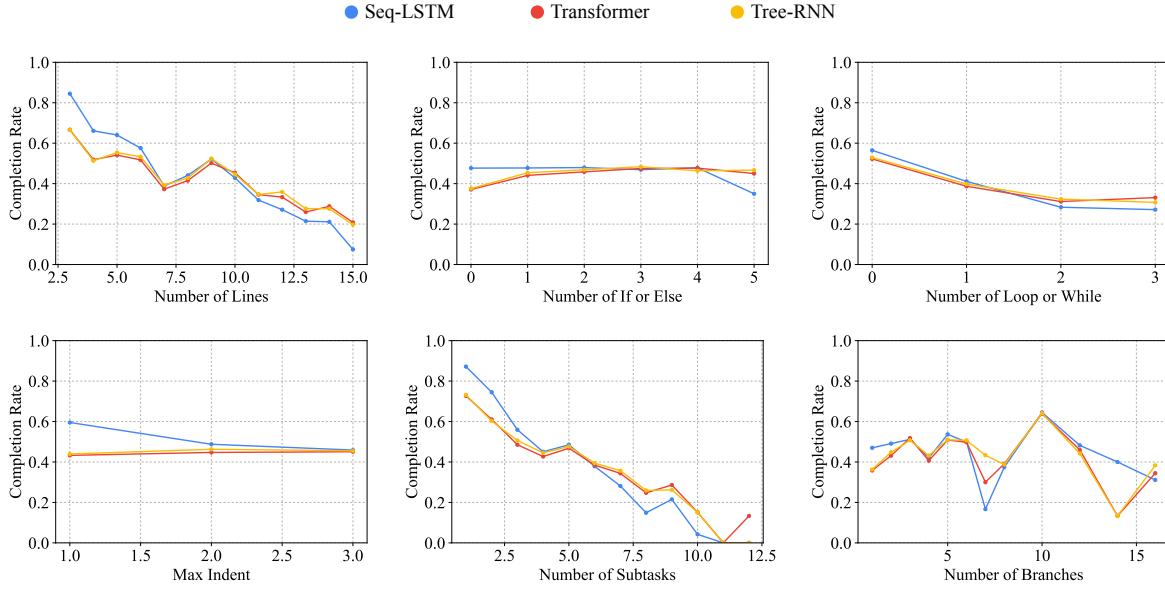


Figure 7.9: Additional analysis on completion rates. The results of executing program instructions on both datasets are used to produce the six plots above. In each plot, each color corresponds to a different model that we propose. For the two rightmost plots, there is a very small number of outliers that extends out of the right boundary of the plot that we omit for visual interpretability reasons. Please note that the use of colors in this figure is not the same as that in Figure 5 of the main paper.

Average time cost of subtasks Continuing from the previous analysis, we show the average time cost of all successful subtask executions in Figure 7.8. As can be seen in the figure, subtasks in `build_bridge`, `mine`, and `sell` categories take relatively smaller number of time steps to complete. In `goto` and `place` categories, the closer to border the subtask requires the agent to reach, the more time consuming it gets for the agent to complete the subtask. This corresponds to the finding in the analysis of first failure rates that subtasks with destinations close to the border are more likely to fail. In other words, the closer to border the agent has to reach, the more likely it is to fail the subtask.

Additional analysis on end-to-end learning models completion rates To conclude failure analysis, we focus on the variation of completion rates of program executions with respect to different conditioning variables. As shown in Figure 7.9, plots showing the trends of completion rates while evaluating with different independent variables show that execution failure is more common in cases when the program

consists of larger number of lines, more loops and while statements, and larger number of subtasks. Note that this subtask count is only a summation of the occurrence counts of each subtask in a program, which does not accurately reflect the number of executions each subtask is being invoked (*i.e.* it does not reflect the repetitive counts when there is a loop).

Meanwhile, the effect of the number of if and else statements and the maximum indent values of programs on completion rates seem to vary across different models. For Seq-LSTM model, having a larger number of if and else statements or having a larger max indent value results in more failures; while for Transformer and Tree-RNN models, having larger values above results instead in fewer failures. This is probably since Transformer and Tree-RNN models are designed in a way that deals with hierarchical structures with jumps in instruction executions better (this point is also mentioned in the main paper). Despite this difference in effects, the overall change in performance when the number of if and else statements and the maximum indent value change is much less significant than that in the previous case.

During our analysis, we also designed an algorithm to calculate an estimate to the number of branches a program has. Here, the number of branches is defined by the number of distinct sets of lines that a program can be executed. For a program without control flows (no if, else-if, else, loop, and while statements), the number of branches is always 1. For if, else-if, and else statements, the exact number of branches these statements incur can be calculated easily. In cases of the loop and while statements, we treat loops as being executed only once and while statements as if statements when we calculate the number of branches. The result shown in the analysis does not reveal a clear trend. We attribute this result to two possibilities – either the metric we create is not accurate enough, or it is not a very suitable metric to be inspected.

#	Language Instructions	Ground Truth Program	Alternative Interpretation
(a)	If there is a river, build a bridge. Repeat the followings 3 times: mine a gold, and if environment has no more than 8 gold, mine iron, and then sell an iron.	<pre>def run(): if is_there[River]: build_bridge() loop(3): mine(Gold) if env[Gold] <= 8: mine(Gold) sell(Iron)</pre>	<pre>def run(): if is_there[River]: build_bridge() loop(3): mine(Gold) if env[Gold] <= 8: mine(Gold) sell(Iron)</pre>
(b)	Place an iron on (7,2) and repeat 4 times, if agent has no more than 9 iron then sell a gold.	<pre>def run(): place(Iron, 7, 2) loop(4): if agent[Iron] <= 9: sell(Gold)</pre>	<pre>def run(): loop(5): place(Iron, 7, 2) if agent[Iron] <= 9: sell(Gold)</pre>
(c)	Mine wood first. If agent has more than 3 iron, mine wood. If there is gold in the environment, place iron at (3,7).	<pre>def run(): mine(Wood) if agent[Iron] >= 4: mine(Wood) if is_there[Gold]: place(Iron, 3, 7)</pre>	<pre>def run(): mine(Wood) if agent[Iron] >= 4: mine(Wood) if is_there[Gold]: place(Iron, 3, 7)</pre>

Figure 7.10: **Exemplar data and language ambiguity.** The goal of the examples above is to show that natural language instructions while being flexible enough to capture the high-level semantics of the task, can be ambiguous in different ways and thus might lead to impaired performance. In example (a), the modifier "repeat the following 3 times" has an unclear scope, resulting in two possible interpretations shown in program format on the right side; in example (b), "repeat 4 times" can be used to modify either the previous part of the description or the latter part of it, resulting in ambiguity; in example (c), the last sentence starting with "If" has unclear scope. In all of the above cases, a model that learns to execute instructions presented in natural language format might fail to execute the instructions successfully because of the ambiguity of the language instructions.

7.7.5.7 Hyperparameters

We use the following hyperparameters to train A2C agents for our model and all the end-to-end learning models: learning rate: 1×10^{-3} , number of environment: 64, number of workers: 64, and number of update roll-out steps: 5.

7.7.5.8 Computational Resources

We train all our models on a single Nvidia Titan-X GPU, in a 40 core Ubuntu 16.04 Linux server.

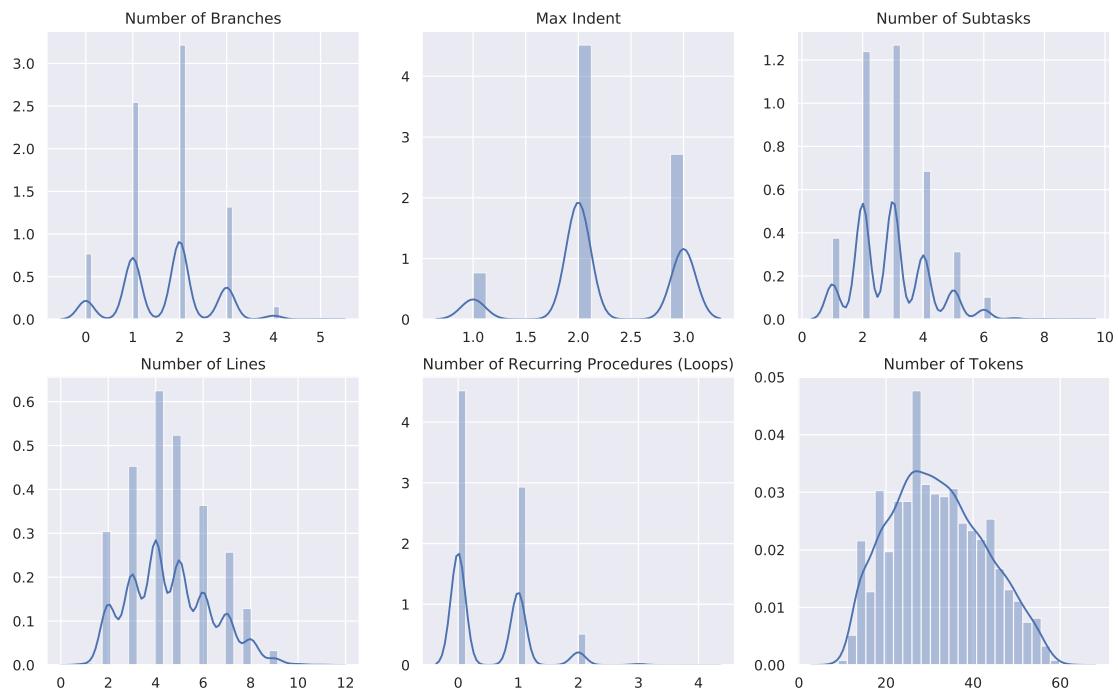


Figure 7.11: Program set statistics for training set (*train*).

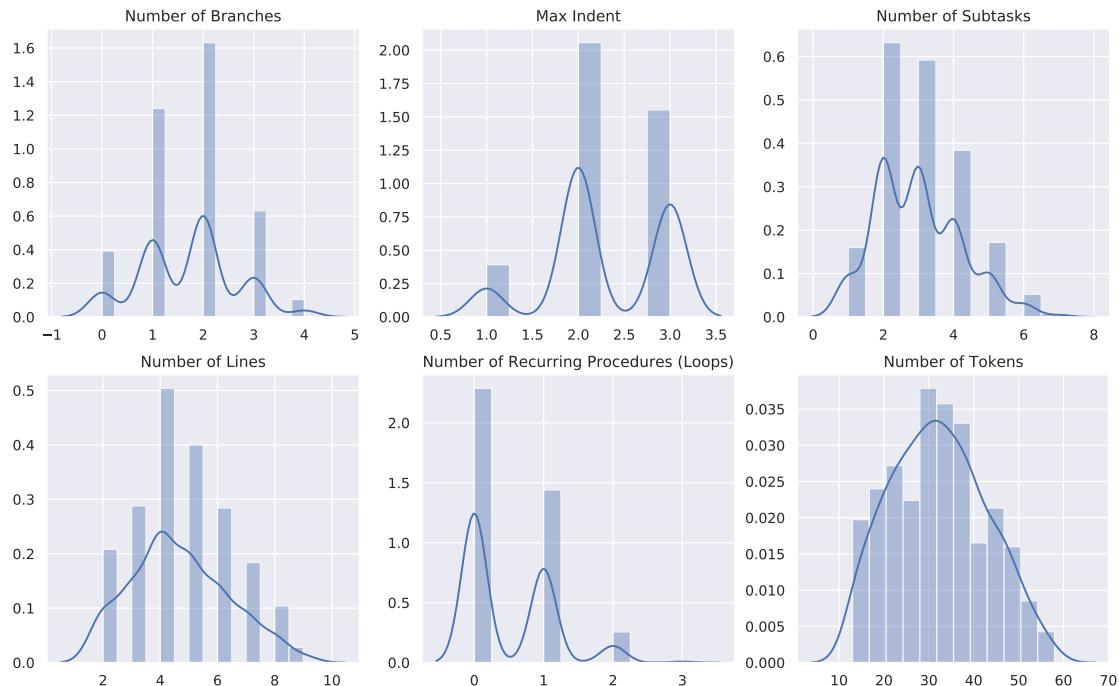


Figure 7.12: Program set statistics for same complexity testing set (*test*).

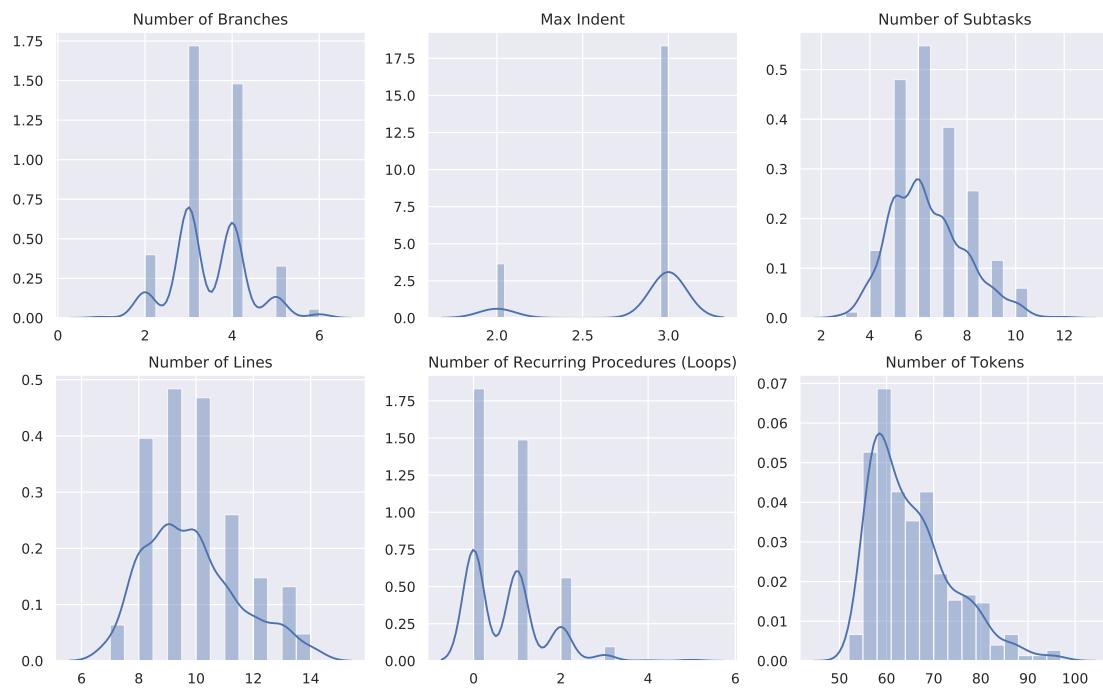


Figure 7.13: Program set statistics for more complex testing set (*test-complex*).

Chapter 8

Learning to Compose Skills

8.1 Introduction

While humans are capable of learning complex tasks by reusing previously learned skills, composing and mastering complex skills are not as trivial as sequentially executing those acquired skills. Instead, it requires a smooth transition between skills since the final pose of one skill may not be appropriate to initiate the following one. For example, scoring in basketball with a quick shot after receiving a ball can be decomposed into catching and shooting. However, it is still difficult for beginners who have learned to catch passes and statically shoot. To master this skill, players must practice adjusting their footwork and body into a comfortable shooting pose after catching a pass.

Can machines similarly learn new and complex tasks by reusing acquired skills and learning transitions between them? Learning to perform composite and long-term tasks from scratch requires extensive exploration and sophisticated reward design, which can introduce undesired behaviors [250]. Thus, instead of employing intricate reward functions and learning from scratch, modular methods sequentially execute acquired skills with a rule-based meta-policy, enabling machines to solve complicated tasks [222, 201, 11]. These modular approaches assume that a task can be clearly decomposed into several subtasks which are smoothly connected to each other. In other words, an ending state of one subtask falls within the set of starting states, *initiation set*, of the next subtask [290]. However, this assumption does not hold in many

continuous control problems where a given skill may be executed in starting states not considered during training or designing and thus, fail to achieve its goal.

To bridge the gap between skills, we propose a *transition policy* which learns to smoothly navigate from an ending state of a skill to suitable initial states of the following skill, as illustrated in Figure 8.1. However, learning a transition policy between skills without reward shaping is difficult as the only available learning signal is the sparse reward for the successful execution of the next skill. Sparse success/failure reward is challenging to learn from due to the temporal credit assignment problem [291] and the lack of information from failing trajectories. To alleviate these problems, we propose a *proximity predictor* which outputs the proximity to the initiation set of the next skill and acts as a dense reward function for the transition policy.

The main contributions of this paper include (1) the concept of learning transition policies to smoothly connect primitive skills; (2) a novel modular framework with transition policies that is able to compose complex skills by reusing existing skills; and (3) a joint training algorithm with the proximity predictor specifically designed for efficiently training transition policies. This framework is suited for learning complex skills that require sequential execution of acquired primitive skills, which are common for humans yet relatively unexplored in robot learning. Our experiments on simulated environments demonstrate that employing transition policies solves complex continuous control tasks which traditional policy gradient methods struggle at.

8.2 Related Work

Learning continuous control of diverse behaviors in locomotion [190, 111, 228] and robotic manipulation [91] is an active research area in reinforcement learning (RL). While some complex tasks can be solved through extensive reward engineering [209], undesired behaviors often emerge [250] when tasks require several different primitive skills. Moreover, training complex skills from scratch is not computationally practical.

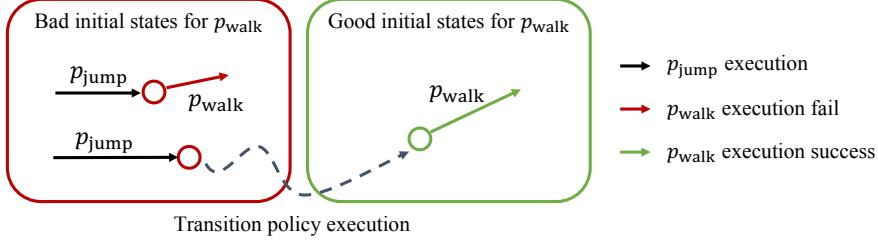


Figure 8.1: Concept of a transition policy. Composing complex skills using primitive skills requires smooth transitions between primitive skills since a following primitive skill might not be robust to ending states of the previous one. In this example, the ending states (red circles) of the primitive policy p_{jump} are not good initial states to execute the following policy p_{walk} . Therefore, executing p_{walk} from these states will fail (red arrow). To smoothly connect the two primitive policies, we propose a transition policy which navigates an agent to suitable initial states for p_{walk} (dashed arrow), leading to a successful execution of p_{walk} (green arrow).

Real-world tasks often require diverse behaviors and longer temporal dependencies. In hierarchical reinforcement learning, the option framework [290] learns meta actions (options), a series of primitive actions over a period of time. Typically, a hierarchical reinforcement learning framework consists of two components: a high-level meta-controller and low-level controllers. A meta-controller determines the order of subtasks to achieve the final goal and chooses corresponding low-level controllers that generate a sequence of primitive actions. Unsupervised approaches to discover meta actions have been proposed [260, 56, 20, 308, 66, 164, 84, 248, 184]. However, to deal with more complex tasks, additional supervision signals [11, 190, 298] or pre-defined low-level controllers [148, 216] are required.

To exploit pre-trained modules as low-level controllers, neural module networks [12] have been proposed, which construct a new network dedicated to a given query using a collection of reusable modules. In the RL domain, a meta-controller is trained to follow instructions [216] and demonstrations [331], and support multi-level hierarchies [99]. In the robotics domain, Pastor et al. [222], Kober et al. [142], and Mülling et al. [201] have proposed a modular approach that learns table tennis by selecting appropriate low-level controllers. On the other hand, Andreas, Klein, and Levine [11] and Frans et al. [84] learn abstract skills while experiencing a distribution of tasks and then solve a new task with the learned primitive skills. However, these modular approaches result in undefined behavior when two skills are not smoothly

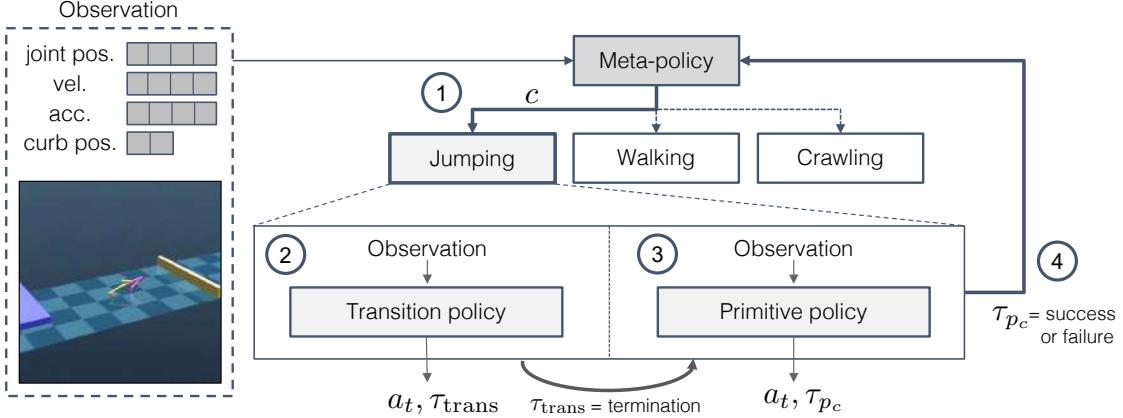


Figure 8.2: **Our modular network augmented with transition policies.** To perform a complex task, our model repeats the following steps: (1) The meta-policy chooses a primitive policy of index c ; (2) The corresponding transition policy helps initiate the chosen primitive policy; (3) The primitive policy executes the skill; and (4) A success or failure signal for the primitive skill is produced.

connected. Our proposed framework aims to bridge this gap by training transition policies in a model-free manner to navigate the agent from unseen states for following skills to suitable initial states.

Deep RL techniques for continuous control demand dense reward signals; otherwise, they suffer from long training time. Instead of manual reward shaping for denser reward, adversarial reinforcement learning [114, 190, 322, 21] employs a discriminator which learns to judge the state or the policy, and the policy takes as rewards the output of the discriminator. While those methods assume ground truth trajectories or goal states are given, our method collects both success and failure trajectories online to train proximity predictors which provide rewards for transition policies.

8.3 Approach

In this paper, we address the problem of solving a complex task that requires sequential composition of primitive skills given only *sparse and binary rewards* (*i.e.* subtask completion reward). The sequential execution of primitive skills fails when two consecutive skills are not smoothly connected. We propose a modular framework with *transition policies* that learn to make transition between one policy to the subsequent policy, and therefore, can exploit the given primitive skills to compose complex skills. To

accelerate training of transition policies, additional networks, *proximity predictors*, are jointly trained to provide *proximity rewards* as intermediate feedback to transition policies. In Section 8.3.2, we describe our framework in details. Next, in Section 8.3.3, we elaborate how transition policies are efficiently trained with induced proximity reward.

8.3.1 Preliminaries

We formulate our problem as a Markov decision process defined by a tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{T}, R, \rho, \gamma\}$ of states, actions, transition probability, reward, initial state distribution, and discount factor. An action distribution of an agent is represented as a policy $\pi_\theta(a_t|s_t)$, where $s_t \in \mathcal{S}$ is a state, $a_t \in \mathcal{A}$ is an action at time t , and θ are the parameters of the policy. An initial state s_0 is randomly sampled from ρ , and then, an agent iteratively takes an action a_t sampled from a policy $\pi_\theta(a_t|s_t)$ and receives a reward r_t until the episode ends. The performance of the agent is evaluated based on a discounted return $R = \sum_{t=0}^{T-1} \gamma^t r_t$, where T is the episode horizon.

8.3.2 Modular Framework with Transition Policies

To learn a new task given primitive skills $\{p_1, p_2, \dots, p_n\}$, we design a modular framework that consists of the following components: a meta-policy, primitive policies, and transition policies. The meta-policy chooses a primitive skill p_c to execute at the beginning and whenever the primitive skill is terminated. Prior to running p_c , the transition policy for p_c is executed to bring the current state to a plausible initial state for p_c , and therefore, p_c can be successfully performed. This procedure is repeated to compose complex skills as illustrated in Figure 8.2 and Algorithm 5.

We denote the meta-policy as $\pi_{meta}(p_c|s)$, where $c \in [1, n]$ is a primitive policy index. The observation of the meta-policy contains the low-level information of primitives and task specifications indicating high-level goals (e.g. moving direction and target object position). For example, a walking primitive only

takes joint information as observation while the meta-policy additionally takes target direction. In this paper, we use a rule-based meta-policy and focus on transitioning between consecutive primitive policies.

Once a primitive skill p_c is chosen to be executed, the agent generates an action $a_t \sim \pi_{p_c}(a|s_t)$ based on the current state s_t . Note that we did not differentiate state spaces for primitive policies because of the simplicity of notations (e.g. the observation of the jumping primitive contains a distance to a curb while that of the walking primitive only has joint pose and velocities). Every primitive policy is required to generate termination signals $\tau_{p_c} \in \{\text{continue}, \text{success}, \text{fail}\}$ to indicate policy completion and whether it believes the execution is successful or not. While our method is agnostic to the form of primitive policies (e.g. rule-based, inverse kinematics), we consider the case of a pre-trained neural network in this paper.

For smooth transitions between primitive policies, we add a transition policy $\pi_{\phi_c}(a|s)$ before executing primitive skill p_c , which guides an agent to p_c 's initiation set, where ϕ_c is the parameters of the transition policy for p_c . Note that the transition policy for p_c is shared across different preceding primitive policies since a successful transition is defined by the success of the following primitive skill p_c . For brevity of notation, we omit the primitive policy index c in the following equations where unambiguous. The transition policy's state and action space are the same as the primitive policy's. The transition policy also learns a termination signal τ_{trans} which indicates transition termination to successfully initiate p_c . Our framework contains one transition policy for each primitive skill, in total n transition policies $\{\pi_{\phi_1}, \pi_{\phi_2}, \dots, \pi_{\phi_n}\}$.

8.3.3 Training Transition Policies

In our framework, transition policies are trained to make the execution of the corresponding following primitive policies successful. During rollouts, transition trajectories are collected and each trajectory can be naively labeled by the success execution of its corresponding primitive policy. Then, transition policies are trained to maximize the average success of the respective primitive policy. In this scenario, by definition, the

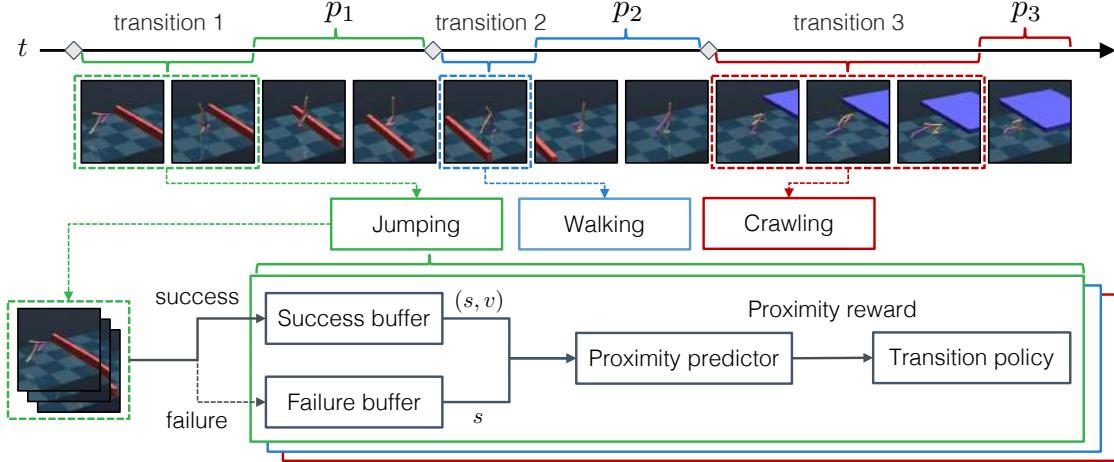


Figure 8.3: **Training of transition policies and proximity predictors.** After executing a primitive policy, a previously performed transition trajectory is labeled and added to a replay buffer based on the execution success. A proximity predictor is trained on states sampled from the two buffers to output the proximity to the initiation set. The predicted proximity serves as a reward to encourage the transition policy to move toward good initial states for the corresponding primitive policy.

only available learning signal for the transition policies is the sparse and binary rewards for the completion of the next task.

To alleviate the sparsity of rewards and maximize the objective of moving to viable initial states for the next primitive, we propose a *proximity predictor* that learns and provides a dense reward, dubbed *proximity reward*, of how close transition states are to the initiation set of the corresponding primitive p_c as shown in Figure 8.3. We denote a proximity predictor as P_{ω_c} which is parameterized by ω_c . We define the proximity of a state as the future discounted proximity, $v = \delta^{step}$, where $step$ is the number of steps required to reach an initiation set of the following primitive policy. The proximity of a state can also be a linearly discounted function such as $v = 1 - \delta \cdot step$. We refer the readers to Appendix Section 8.6 for comparison of two proximity functions.

The proximity predictor is trained to minimize a mean squared error of proximity prediction:

$$L_P(\omega, \mathcal{B}^S, \mathcal{B}^F) = \frac{1}{2} \mathbb{E}_{(s,v) \sim \mathcal{B}^S} [(P_\omega(s) - v)^2] + \frac{1}{2} \mathbb{E}_{s \sim \mathcal{B}^F} [P_\omega(s)^2], \quad (8.1)$$

where \mathcal{B}^S and \mathcal{B}^F are collections of states from success and failure transition trajectories, respectively. To estimate the proximity to an initiation set, \mathcal{B}^S contains not only the state that directly leads to the success of the following primitive policy, but also the intermediate states of the successful trajectories with its proximity. By minimizing this objective, given a state, the proximity predictor is learned to predict 1 if the state is in the initiation set, a value that is between 0 and 1 if the state leads the agent to end up with a desired initial states, and 0 when the state leads to a failure.

The goal of a transition policy is to get close to an initiation set which can be formulated as seeking a state s predicted to be in the initiation set by the proximity predictor (*i.e.* $P_\omega(s)$ is close to 1). To achieve this goal, the transition policy learns to maximize proximity prediction at the ending state of the transition trajectory $P_\omega(s_T)$. In addition to providing reward at the end, we also use the increase of predicted proximity to the initiation set, $P_\omega(s_{t+1}) - P_\omega(s_t)$, at every timestep as a reward, dubbed *proximity reward*, to create a denser reward. The transition policy is trained to maximize the expected discounted return:

$$R_{\text{trans}}(\phi) = \mathbb{E}_{(s_0, s_1, \dots, s_T) \sim \pi_\phi} \left[\gamma^T P_\omega(s_T) + \sum_{t=0}^{T-1} \gamma^t (P_\omega(s_{t+1}) - P_\omega(s_t)) \right]. \quad (8.2)$$

However, in general skill learning scenarios, ground truth states (\mathcal{B}^S and \mathcal{B}^F) for training proximity predictors are not available. Hence, the training data for a proximity predictor is obtained online during training its corresponding transition policy. Specifically, we label the states in a transition trajectory as success or failure based on whether the following primitive is successfully executed or not, and add them into the corresponding buffers \mathcal{B}^S or \mathcal{B}^F , respectively. As stated in Algorithm 4, we train transition policies and proximity predictors by alternating between an Adam [140] gradient step on ω to minimize Equation (8.1) with respect to P_ω and a PPO [265] step on ϕ to maximize Equation (8.2) with respect to π_ϕ . We refer readers to Appendix Section 8.6 for further details on training.

In summary, we propose to compose complex skills with transition policies that enable smooth transition between previously acquired primitive policies. Specifically, we propose to reward transition policies based on how close the current state is to suitable initial states of the subsequent policy (*i.e.* initiation set). To provide the proximity of a state, we collect failing and successful trajectories on the fly and train a proximity predictor to predict the proximity.

Utilizing the learned proximity predictors and proximity rewards for training transition policies is beneficial in the following perspectives: (1) the dense rewards speed up transition policy training by differentiating failing states from states in a successful trajectory; and (2) the joint training mechanism prevents a transition policy from getting stuck in local optima. Whenever a transition policy gets into a local optimum (*i.e.* fails the following skill with a high proximity reward), the proximity predictor learns to lower the proximity for the failing transition as those states are added to its failure buffer, escaping the local optimum.

8.4 Experiments

We conducted experiments on two classes of continuous control tasks: robotic manipulation and locomotion. To illustrate the potential of the proposed framework, modular framework with Transition Policies (TP), we designed a set of complex tasks that require agents to utilize diverse primitive skills which are not optimized for smooth composition. All of our environments are simulated in the MuJoCo physics engine [299].

8.4.1 Baselines

We evaluate our method to answer how transition policies benefit complex task learning and how joint training with proximity predictors boosts training of transition policies. To investigate the impact of the transition policy, we compared policies learned from dense rewards with our modular framework that only learns from sparse and binary rewards (*i.e.* subtask completion rewards). Moreover, we conducted ablation

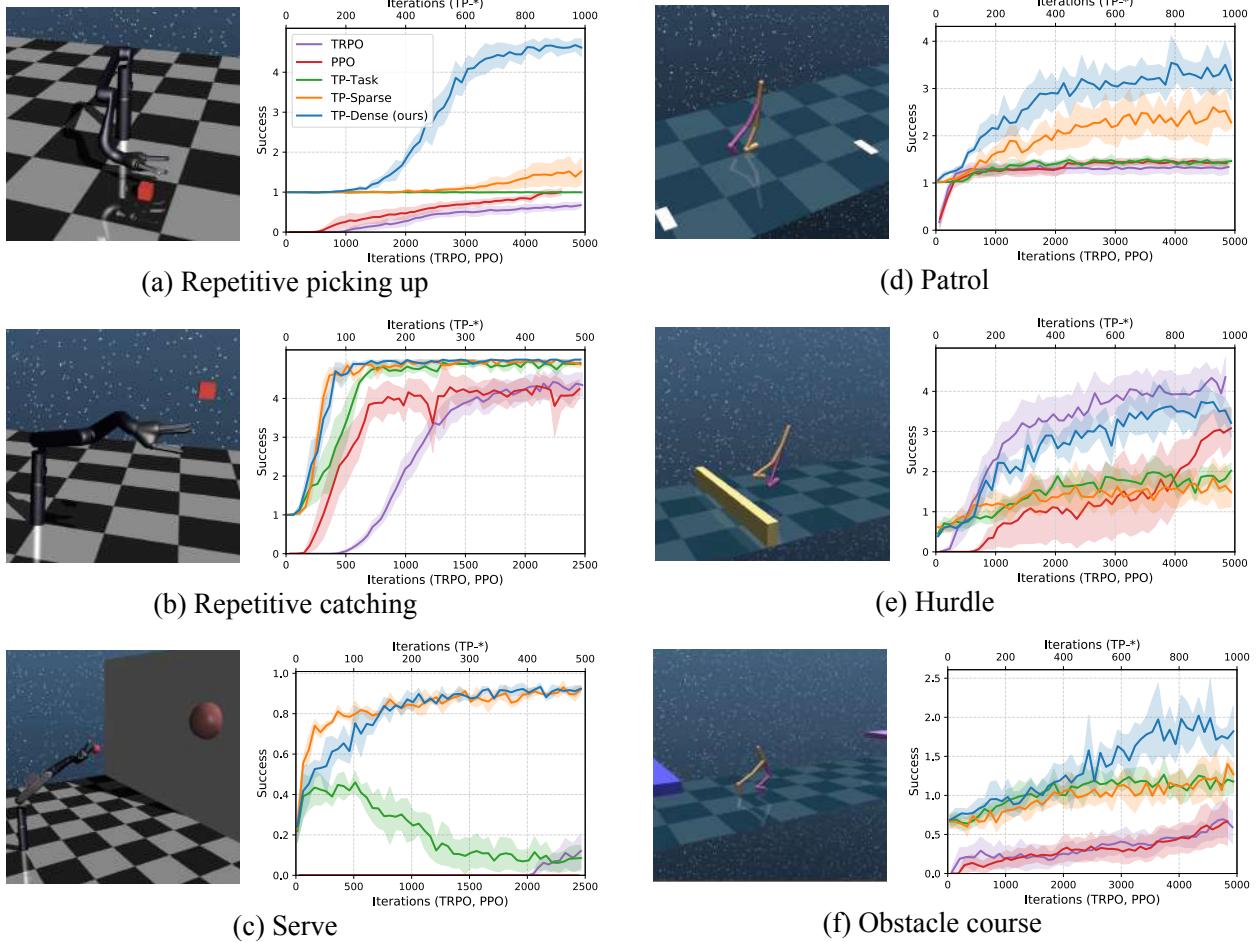


Figure 8.4: Tasks and success count curves of our model (blue), TRPO (purple), PPO (magenta), and transition policies (TP) trained on task reward (green) and sparse proximity reward (yellow). Our model achieves the best performance and convergence time. Note that TRPO and PPO are trained 5 times longer than ours with dense rewards since TRPO and PPO do not have primitive skills and learn from scratch. In the success count curves, different temporal scales are used for TRPO and PPO (bottom x-axis) and ours (top x-axis).

studies to dissect each component in the training method of transition polices. To answer these questions, we compare the following methods:

- **Trust Region Policy Optimization with dense reward (TRPO)** represents a state-of-the-art policy gradient method [264], which we use for the standard RL comparison.
- **Proximal Policy Optimization with dense reward (PPO)** is another state-of-the-art policy gradient method [265], which is more stable than TRPO with smaller batch sizes.

- **Without transition policies (Without-TP)** sequentially executes primitive policies without transition policies and has no learnable components.
- **Transition policies trained on task rewards (TP-Task)** represents a modular network augmented with transition policies learned from the sparse and binary reward (*i.e.* subtask completion reward), whereas our model learns from the dense proximity reward.
- **Transition policies trained on sparse proximity rewards (TP-Sparse)** is a variant of our model which has the proximity reward only at the end of the transition trajectory. In contrast, our model learns from dense proximity rewards generated every timestep.
- **Transition policies trained on dense proximity rewards (TP-Dense, Ours)** is our final model where transition policies learn from dense proximity rewards.

Initially, we tried comparing baseline methods with our method using only sparse and binary rewards. However, the baselines could not solve any of the tasks due to the complexity and sparse reward of the environments. To provide more competitive comparisons, we engineer dense rewards for baselines (TRPO and PPO) to boost their performance and give baselines 5 times longer training times. We show that transitions with sparse rewards can compete with and even outperform baselines learning from dense rewards. As the performance of TRPO and PPO varies significantly between runs, we train each task with 3 different random seeds and report mean and standard deviation in Figure 8.4.

8.4.2 Robotic Manipulation

For robotic manipulation, we simulate a Kinova Jaco, a 9 DoF robotic arm with 3 fingers. The agent receives full state information, including the absolute location of external objects. The agent uses joint torque control to perform actions. The results are shown in Figure 8.4 and Table 8.1.

Pre-trained primitives. There are four pre-trained primitives available: *Picking up*, *Catching*, *Tossing*, and *Hitting*. *Picking up* requires the robotic arm to pick up a small block, which is randomly placed on the

Table 8.1: Success count for robotic manipulation, comparing our method against baselines with or without transition policies (TP). Our method achieves the best performance over both RL baselines and the ablated variants. Each entry in the table represents average success count and standard deviation over 50 runs with 3 random seeds.

	Reward	Repetitive picking up	Repetitive catching	Serve
TRPO	dense	0.69 ± 0.46	4.54 ± 1.21	0.32 ± 0.47
PPO	dense	0.95 ± 0.53	4.26 ± 1.63	0.00 ± 0.00
Without TP	sparse	0.99 ± 0.08	1.00 ± 0.00	0.11 ± 0.32
TP-Task	sparse	0.99 ± 0.08	4.87 ± 0.58	0.05 ± 0.21
TP-Sparse	sparse	1.52 ± 1.12	4.88 ± 0.59	0.92 ± 0.27
TP-Dense (ours)	sparse	4.84 ± 0.63	4.97 ± 0.33	0.92 ± 0.27

table. If the box is not picked up after a certain amount of time, the agent fails. *Catching* learns to catch a block that is thrown towards the arm with random initial position and velocity. The agent fails if it does not catch and stably hold the box for a certain amount of time. *Tossing* requires the robot to pick up a box, toss it vertically in the air, and land the box at a specified position. *Hitting* requires the robot to hit a box dropped overhead at a target ball.

Repetitive picking up. The *Repetitive picking up* task requires the agent to complete the *Picking up* task 5 times. After each successful pick, the box disappears and a new box will be placed randomly on the table again. Our model achieves the best performance and converges the fastest by learning from the proposed proximity reward. With our dense proximity reward at every transition step, we alleviate credit assignment when compared to providing a sparse proximity reward (TP-Sparse) or using a sparse task reward (TP-Task). Conversely, TRPO and PPO with dense rewards take significantly longer to learn and is unable to pick up the second box as the ending pose after the first picking up is too unstable to initialize the next picking up.

Repetitive catching. Similar to *Repetitive picking up*, the *Repetitive catching* task requires the agent to catch boxes consecutively up to 5 times. In this task, other than the modular network without a transition

policy, all baselines are able to eventually learn while our model still learns the fastest. We believe this is because the *Catching* primitive policy has a larger initiation set and therefore, the sparse reward problem is less severe since random exploration is able to succeed with a higher chance.

Serve. Inspired by tennis, *Serve* requires the robot to toss the ball and hit it at a target. Even with an extensively engineered reward, TRPO and PPO baselines fail to learn because *Hitting* is not able to learn to cover all terminal states of *Tossing* (*i.e.* a set of initial states for *Hitting* is large which demands longer training time). In contrast, learning to recover from *Tossing*'s ending states to *Hitting*'s initiation set is easier for exploration (11% of *Tossing*'s ending states are covered by *Hitting*'s initiation set as can be seen in Table 8.1), which reduces the complexity of the task. Thus, our method and the sparse proximity reward baseline are both able to solve it. However, the ablated variant trained on task reward shows high success rates at the beginning of training and collapses after 100 iterations. The performance drops because the transition policy tries to solve failure cases by increasing the transition length and it reaches to a point that it hardly gets reward. This result shows that once the policy falls into local optima, it is not able to escape because the policy will never get a sparse task reward. On the other hand, our method is robust to local optima since the jointly learned dense proximity reward provides a learning signal to an agent even though it cannot get a task reward.

8.4.3 Locomotion

For locomotion, we simulate a 9 DoF planar (2D) bipedal walker. The observation of the agent includes joint position, rotation, and velocity. When the agent needs to interact with objects in the environment, we provide additional input such as distance to the curb and ceiling in front of the agent. The agent uses joint torque control to perform actions. The results are shown in Figure 8.4 and Table 8.2.

Pre-trained primitives. *Forward* and *Backward* require the walker to walk forward and backward with a certain velocity, respectively. *Balancing* requires the walker to robustly stand still under the random

Table 8.2: Success count for locomotion, comparing our method against baselines with or without transition policies (TP). Our method outperforms all baselines in *Patrol* and *Obstacle course*. In *Hurdle*, the reward function for TRPO was extensively engineered, which is not directly comparable to our method. Our method outperforms baselines learning from sparse reward, showing the effectiveness of the proposed proximity predictor. Each entry in the table represents average success count and standard deviation over 50 runs with 3 random seeds.

	Reward	Patrol	Hurdle	Obstacle course
TRPO	dense	1.37 ± 0.52	4.13 ± 1.54	0.98 ± 1.09
PPO	dense	1.53 ± 0.53	2.87 ± 1.92	0.85 ± 1.07
Without TP	sparse	1.02 ± 0.14	0.49 ± 0.75	0.72 ± 0.72
TP-Task	sparse	1.69 ± 0.63	1.73 ± 1.28	1.08 ± 0.78
TP-Sparse	sparse	2.51 ± 1.26	1.47 ± 1.53	1.32 ± 0.99
TP-Dense (Ours)	sparse	3.33 ± 1.38	$3.14 \pm 1.69^*$	1.90 ± 1.45

external forces. *Jumping* requires the walker jump over a randomly located curb and land safely. *Crawling* requires the walker to crawl under a ceiling. In all the aforementioned scenarios, the walker fails when the height of the walker is lower than a threshold.

Patrol (Forward and backward). The *Patrol* task involves walking forward and backward toward goal points on either side and balancing in between to smoothly change its direction. As illustrated in Figure 8.4, our method consistently outperforms TRPO, PPO, and ablated baselines in stably walking forward and transitioning to walk backward. The agent trained with dense rewards is not able to consistently switch directions, whereas our model can utilize previously learned primitives including *Balancing* to stabilize a reversal in velocity.

Hurdle (Walking forward and jumping). The *Hurdle* task requires the agent to walk forward and jump across curbs, which requires a transition between walking and jumping as well as landing the jump to walking forward. As shown in Figure 8.4, our method outperforms the sparse reward baselines, showing the efficiency our proposed proximity reward. While TRPO with dense rewards can learn this task as well, it requires dense rewards consisting of eight different components to collectively enable TRPO to learn the

task. It can be considered as learning both primitive skills and transition between skills from dense rewards. However, the main focus of this paper is to learn a complex task by reusing acquired skills, avoiding an extensive reward design.

Obstacle Course (Walking forward, jumping, and crawling). *Obstacle Course* is the most difficult among the locomotion tasks, where the walker must walk forward, jump across curbs, and crawl underneath ceilings. It requires three different behaviors and transitions between two very different primitive skills: crawling and jumping. Since the task requires significantly different behaviors that are hard to transition between, TRPO fails to learn the task and only tries to crawl toward the curb without attempting to jump. In contrast, our method learns to transition between all pairs of primitive skills and often succeeds in crossing multiple obstacles.

8.4.4 Ablation Study

We conducted additional experiments to understand the contribution of transition policies, proximity predictors, and dense proximity rewards. The modular framework without transition policies (Without-TP) tends to fail the execution of the second skill since the second skill is not trained to cover ending states of the first skill. Especially, in continuous control making a primitive skill that can cover all possible states is very challenging. Transition policies trained from task completion reward (TP-Task) and sparse proximity reward (TP-Sparse) learn to connect consecutive primitives slower because sparse reward is hard to learn from due to the credit assignment problem. On the other hand, our model alleviates the credit assignment problem and learns quickly by giving predicted proximity reward for every transition state-action pair.

8.4.5 Training of Transition Policy and Proximity Predictor

To investigate how transition policies learn to solve the tasks, we present the lengths of transition trajectories and the obtained proximity rewards during training in Figure 8.5. For manipulation, we show the results of

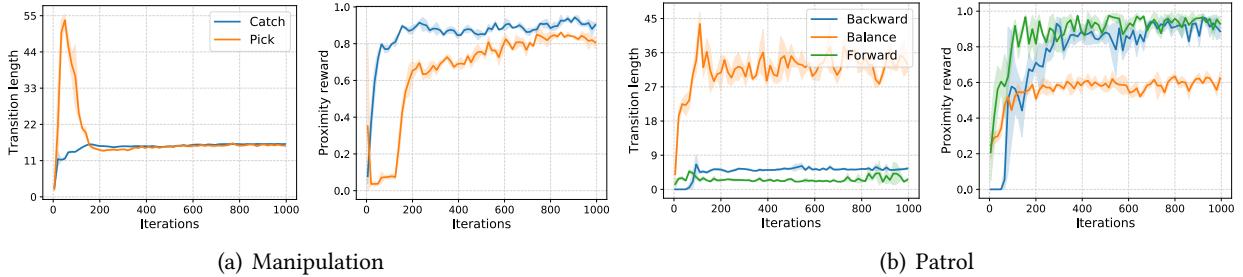


Figure 8.5: Average transition length and average proximity reward of transition trajectories over training on *Manipulation* (left) and *Patrol* (right).

Repetitive picking up and *Repetitive catching*. For locomotion, we show *Patrol* with three different transition policies.

The transition policy quickly learns to maximize the proximity reward regardless of the accuracy of the proximity predictor. All the transition policies increase the length while exploring in the beginning, especially for *picking up* (55 steps) and *balance* (45 steps). This is because a randomly initialized proximity predictor outputs high proximity for unseen states and a transition policy tries to get a high reward by visiting these states. However, as these failing initial states with high proximity are collected in the failure buffers, the proximity predictor lowers their proximity and the transition policy learns to avoid them. In other words, the transition policy will end up seeking successful states. As transition policies learn to transition to the following skills, the length decreases to get higher proximity rewards earlier.

8.4.6 Visualizing Transition Trajectory

Figure 8.6(a) shows two transition trajectories (from s_0 to t_0 and s_1 to t_1) and two-dimensional PCA embedding of the ending states (blue) and initiation states (red) of the *Picking up* primitive. A transition policy starts from states s_0 and s_1 where the previous *Picking up* primitive is terminated. As can be seen in Figure 8.6(a), the proximity predictor outputs small values for s_0 and s_1 since they are far from the initiation set of *Picking up* primitive. Trajectories in the figure show that as the transition policy moves

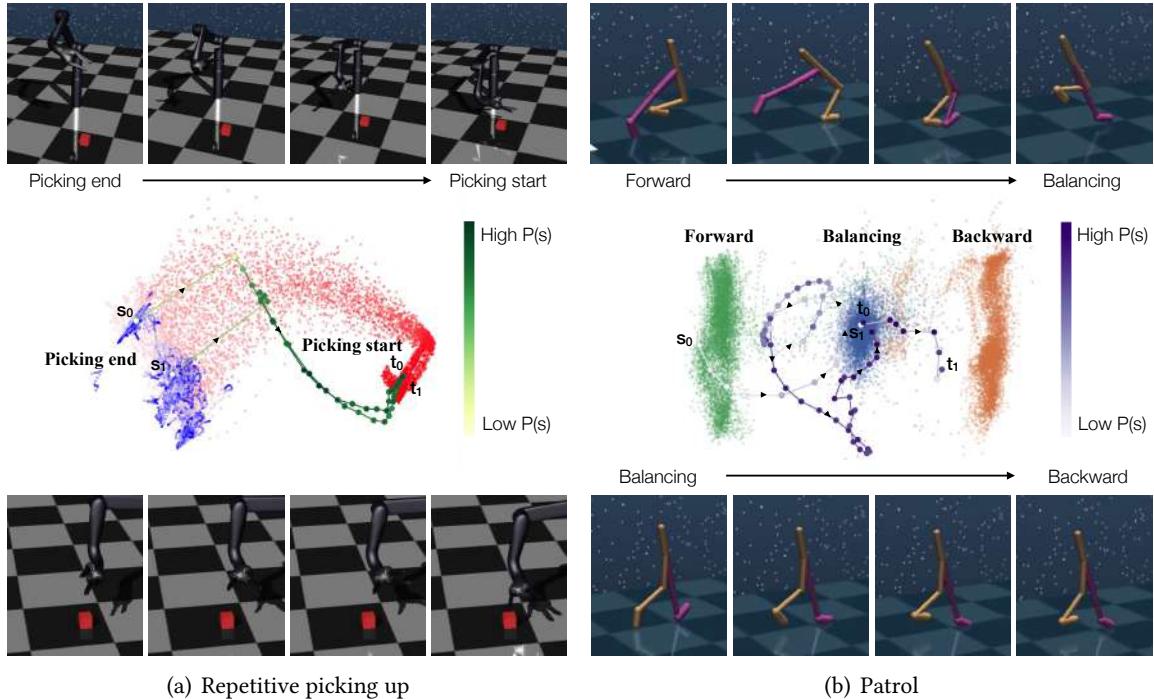


Figure 8.6: Visualization of transition trajectories of (a) *Repetitive picking up* and (b) *Patrol*. TOP AND BOTTOM ROWS: contain rendered frames of transition trajectories. MIDDLE ROW: contains states extracted from each primitive skill execution projected onto PCA space. The dots connected with lines are extracted from the same transition trajectory, where the marker color indicates the proximity prediction $P(s)$. A higher $P(s)$ value indicates proximity to states suitable for initializing the next primitive skill. LEFT: two picking up transition trajectories demonstrate that the transition policy learns to navigate from terminate states s_0 and s_1 to t_0 and t_1 . RIGHT: the forward to balance transition moves between the forward and balance state distributions and the balance to backward transition moves from the balancing states close to the backward states.

toward states with higher proximity, and finally ends up with states t_0 and t_1 which are in the initiation set of the primitive policy.

Figure 8.6(b) illustrates PCA embeddings of initiation sets of three primitive skills, *Forward* (green), *Backward* (orange), and *Balancing* (blue). A transition from *Forward* to *Balancing* has very long trajectory, but predicted proximity helps the transition policy to reach to an initiation state t_0 . On the other hand, transitioning between *Balancing* and *Backward* only requires 7 steps.

8.5 Conclusion

In this work, we propose a modular framework with transition policies to empower reinforcement learning agents to learn complex tasks with sparse reward by utilizing prior knowledge. Specifically, we formulate the problem as executing existing primitive skills while smoothly transitioning between primitive skills. To learn transition policies in a sparse reward setting, we propose a proximity predictor which generates dense reward signals and jointly train transition policies and proximity predictors. Our experimental results on robotic manipulation and locomotion tasks demonstrate the effectiveness of employing transition policies. The proposed framework solves complex tasks without reward shaping and outperforms baseline RL algorithms and other ablated baselines.

There are many future directions to investigate. Our method is designed to focus on acquiring transition policies that connect a given set of primitive policies under the predefined meta-policy. We believe that joint learning of a meta-policy and transition policies on a new task would make our framework more flexible. Moreover, we made an assumption that a successful transition between two consecutive policies should be achievable by random exploration. To alleviate the exploration problem with sparse rewards, our transition policy training can incorporate exploration methods such as count-based exploration bonuses [27, 187] and curiosity-driven intrinsic reward [224]. We also assume our primitive policies return a signal that indicates whether the execution should be terminated or not, similar to Kulkarni et al. [148], Oh et al. [216], and Le et al. [156]. Learning to assess the successful termination of primitive policies together with learning transition policies is a promising future direction.

8.6 Appendix

8.6.1 Acquiring Primitive Policies

The modular framework proposed in this paper allows a primitive policy to be any of a pre-trained neural network, inverse kinematics module, or hard-coded policy. In this paper, we use neural networks trained with TRPO [264] on dedicated environments as primitive policies (see Section 8.6.3 for the details of environments and reward functions). All policy networks we used consists of 2 layers of 32 hidden units with tanh nonlinearities and predicts the mean and standard deviation of a Gaussian distribution over an action space. We trained all primitive policies until the total return converged (up to 10,000 iterations).

Given a state, a primitive policy outputs an *action* as well as a *termination signal* indicating whether the execution is done and if the skill was successfully performed (see Section 8.6.3 for details on primitive skills and termination conditions).

8.6.2 Training Details

8.6.2.1 Implementation Details

For the TRPO and PPO implementation, we used OpenAI baselines [64] with default hyperparameters including learning rate, KL penalty, and entropy coefficients unless specified below.

Hyperparameters	Transition policy	Proximity predictor	Primitive policy	TRPO	PPO
Learning rate	1e-4	1e-4	1e-3 (for critic)	1e-3 (for critic)	1e-4
# Mini-batch	150	150	32	150	150
Mini-batch size	64	64	64	64	64
Learning rate decay	no	no	no	no	linear decay

Table 8.3: Hyperparameter values for transition policy, proximity predictor, and primitive policy as well as TRPO and PPO baselines.

For all networks, we use the Adam optimizer with mini-batch size of 64. We use 4 workers for rollout and parameter update. The size of rollout for each update is 10,000 steps. We limit the maximum length of a transition trajectory as 100.

8.6.2.2 Replay Buffers

A success buffer \mathcal{B}^S contains states and their proximity to the corresponding initiation set in successful transitions. On the other hand, a failure buffer \mathcal{B}^F contains states in failure transitions. Both the two buffers are FIFO (*i.e.* new items are added on one end and once a buffer is full, a corresponding number of items are discarded from the opposite end). For all experiments, we use buffers, \mathcal{B}^S and \mathcal{B}^F , with a capacity of one million states.

For efficient training of the proximity predictors, we collect successful trajectories of primitive skills which can be sampled during the training of primitive skills. We run 1,000 episodes for each primitive and put the first 10 - 20% in trajectories into the success buffer as an initiation set. While initiation sets can be discovered via random exploration, we found that this initialization of success buffers improves the efficiency of training by providing initial training data for the proximity predictors.

8.6.2.3 Proximity Reward

Transition policies receive rewards based on the outputs of proximity predictors. Before computing the reward at every time step, we clip the output of the proximity predictor P by $\text{clip}(P(s), 0, 1)$ which indicates how close the state s is to the initiation set of the following primitive (higher values correspond to closer states). We define the proximity of a state to an initiation set as an exponentially discounted function δ^{step} , where $step$ is the shortest number of timesteps required to get to a state in the initiation set. We use $\delta = 0.95$ for all experiments. To make the reward denser, for every timestep t , we provide the increase in proximity, $P(s_{t+1}) - P(s_t)$, as a reward for transition policy.

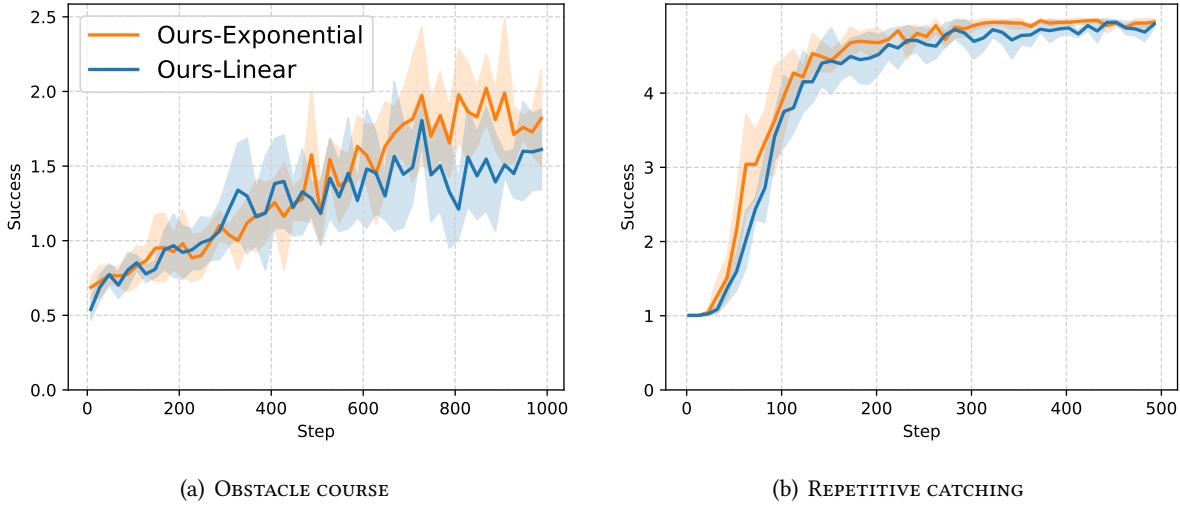


Figure 8.7: Success count curves of our model with exponentially discounted proximity function and linearly discounted proximity function over training on *Obstacle course* (left) and *Repetitive catching* (right).

Using a linearly discounted proximity function, $1 - \delta \cdot step$, is also a valid choice. We compare the two proximity functions on a manipulation task (*Repetitive catching*) and a locomotion task (*Obstacle course*), as shown in Figure 8.7, where δ for exponential decay and linear decay are 0.95 and 0.01, respectively. The results demonstrate that our model is able to learn well with both proximity functions and they perform similarly.

Originally, we opted for the exponential proximity function with the intuition that the faster initial decay near the initiation set would help the policy discriminate successful states from failing states near the initiation set. Also, in our experiments, as we use 0.95 as a decaying factor, the proximity is still reasonably large (e.g., 0.35 for 20 time-steps and 0.07 for 50 time-steps). In this paper, we use the exponential proximity function for all experiments.

8.6.2.4 Proximity Predictor

A proximity predictor takes a state as input which includes joint state information, joint acceleration, and any task specification, such as ceiling and curb information. A proximity predictor consists of 2 fully connected layers of 96 hidden units with ReLU nonlinearities and predicts the proximity to the initiation

set based on the states sampled from the success and failure buffers. Each training iteration consists of 10 epochs over a batch size of 64 and use a learning rate of 10^{-4} . The predictor optimizes the loss in Equation (8.1), similar to the LSGAN loss [186].

8.6.2.5 Transition Policies

An observation space of a transition policy consists of joint state information and joint acceleration. A transition policy consists of 2 fully connected layers of 32 hidden units with tanh nonlinearities and predicts the mean and standard deviation of a Gaussian distribution over an action space. A 2-way softmax layer is followed by the last fully connected layer to predict whether to terminate the current transition or not. We train all transition policies using PPO [265] since PPO is robust on smaller batch sizes and the transition states collected for each update is much smaller than the size of a rollout. Each training iteration consists of 5 epochs over a batch.

Algorithm 4 TRAIN

- 1: **Input:** Primitive polices $\{\pi_{p_1}, \dots, \pi_{p_n}\}$.
 - 2: Initialize success buffers $\{\mathcal{B}_1^S, \dots, \mathcal{B}_n^S\}$ with successful trajectories of primitive policies.
 - 3: Initialize failure buffers $\{\mathcal{B}_1^F, \dots, \mathcal{B}_n^F\}$.
 - 4: Randomly initialize parameters of transition policies $\{\phi_1, \dots, \phi_n\}$ and proximity predictors $\{\omega_1, \dots, \omega_n\}$.
 - 5: **repeat**
 - 6: Initialize rollout buffers $\{\mathcal{R}_1, \dots, \mathcal{R}_n\}$.
 - 7: Collect trajectories using ROLLOUT.
 - 8: **for** $i = 1$ **to** n **do**
 - 9: Update P_{ω_i} to minimize Equation (8.1) using \mathcal{B}_i^S and \mathcal{B}_i^F .
 - 10: Update π_{ϕ_i} to maximize Equation (8.2) using \mathcal{R}_i .
 - 11: **end for**
 - 12: **until** convergence
-

Algorithm 5 ROLLOUT

- 1: **Input:** Meta policy π_{meta} , primitive policies $\{\pi_{p_1}, \dots, \pi_{p_n}\}$, transition policies $\{\pi_{\phi_1}, \dots, \pi_{\phi_n}\}$, and proximity predictors $\{P_{\omega_1}, \dots, P_{\omega_n}\}$.
- 2: Initialize an episode and receive initial state s_0 .
- 3: $t \leftarrow 0$
- 4: **while** episode is not terminated **do**
- 5: $c \sim \pi_{\text{meta}}(s_t)$
- 6: Initialize a rollout buffer \mathcal{B} .
- 7: **while** episode is not terminated **do**
- 8: $a_t, \tau_{\text{trans}} \sim \pi_{\phi_c}(s_t)$
- 9: Terminate the transition policy if $\tau_{\text{trans}} = \text{terminate}$.
- 10: $s_{t+1}, \tau_{\text{env}} \leftarrow \text{ENV}(s_t, a_t)$
- 11: $r_t \leftarrow P_{\omega_c}(s_{t+1}) - P_{\omega_c}(s_t)$
- 12: Store $(s_t, a_t, r_t, \tau_{\text{env}}, s_{t+1})$ in \mathcal{B}
- 13: $t \leftarrow t + 1$
- 14: **end while**
- 15: **while** episode is not terminated **do**
- 16: $a_t, \tau_{p_c} \sim \pi_{p_c}(s_t)$
- 17: Terminate the primitive policy if $\tau_{p_c} \neq \text{continue}$.
- 18: $s_{t+1}, \tau_{\text{env}} \leftarrow \text{ENV}(s_t, a_t)$
- 19: $t \leftarrow t + 1$
- 20: **end while**
- 21: Compute the discounted proximity v of each state s in \mathcal{B} .
- 22: Add pairs of (s, v) to \mathcal{B}_c^S or \mathcal{B}_c^F according to τ_{p_c} .
- 23: Add \mathcal{B} to the rollout buffer \mathcal{R}_c .
- 24: **end while**

8.6.2.6 Scalability

Each sub-policy requires its corresponding transition policy, proximity predictor, and two buffers. Hence, both the time and memory complexities of our method are linearly dependent on the number of sub-policies. The memory overhead is affordable since a transition policy (2 layers of 32 hidden units), a proximity predictor (2 layers of 96 hidden units), and replay buffers (1M states) are small.

8.6.3 Environment Descriptions

For every task, we add a control penalty, $-0.001 * \|a\|^2$, to regularize the magnitude of actions where a is a torque action performed by an agent. Note that all measures are in meters, and we omit the measures here for clarity of the presentation.

8.6.3.1 Robotic Manipulation

In object manipulation tasks, a 9-DOF Jaco robotic arm^{*} is used as an agent and a cube with the side length 0.06 m is used as a target object. We follow the tasks and environment settings proposed in Ghosh et al. [91]. The observation consists of the position of the base of the Jaco arm, joint angles, angular velocities as well as the position, rotation, velocity, and angular velocity of the cube. The action space is a torque control on 9 joints.

Reward Design and Termination Condition

Picking up: In the *Picking up* task, the position of the box is randomly initialized within a square region of size $0.1 \text{ m} \times 0.1 \text{ m}$ with a center $(0.5, 0.2)$. There is an initial guide reward to guide the arm to the box. There is also an over reward to guide the hand directly over the box. When the arm is not picking up the box, there is a pick reward to incentivize the arm to pick the box up. There is an additional hold reward that makes the arm hold the box in place after picking up. Finally, there is a success reward given after the

^{*}<http://www.mujoco.org/forum/index.php?resources/kinova-arms.12/>

arm has held the box for 50 frames. The success reward is scaled with number of timesteps to encourage the arm to succeed as quickly as possible.

$$R(s) = \lambda_{guide} \cdot \mathbf{1}_{\text{Box not picked and Box on ground}} + \lambda_{pick} \cdot \mathbf{1}_{\text{Box in hand and not picked}} + \lambda_{hold} \cdot \mathbf{1}_{\text{Box picked and near hold point}}$$

$$\lambda_{guide} = 2, \lambda_{pick} = 100, \lambda_{hold} = 0.1$$

Catching: The position of the box is initialized at (0, 2.0, 1.5) and the directional force of size 110 is applied to throw the box toward the agent with randomness (0.1 m × 0.1 m).

$$R(s) = \mathbf{1}_{\text{Box in air and Box within 0.06 of Jaco end-effector}}$$

Tossing: The box is randomly initialized on the ground at (0.4, 0.3, 0.05) within a 0.005 × 0.005 square region. A guide reward is given to guide the arm to the top of the box. A pick reward is then given to lift the box up to a specified release height. A release reward is given if the box is no longer in the hand. A stable reward is given to minimize variation in the box's x and y direction. An up reward is given while the ball is traveling upwards in air, up until the box hits a specified z height. Finally, a success reward +100 is given based on the landing position of the box and the specified landing position.

Hitting: The box is randomly initialized overhead the arm at (0.4, 0.3, 1.2) within a 0.005 × 0.005 m square region. The box falls and the arm is given a hit reward +10 for hitting the box. Once the box has been hit, a target reward is given based on how close the box is to the target.

Repetitive picking up: The *Repetitive picking up* task has two reward variants. The sparse version gives a reward +1 for every successful pick. The dense reward version gives a guide reward to the box after each successful pick following the reward for the *Picking up* task.

Repetitive catching: The *Repetitive catching* task gives a reward +1 for every successful catch. For dense reward, it uses the same reward function with that of the *Catching* task.

Serve: The *Serve* task gives a toss reward +1 for a successful toss and a target reward +1 for successfully hitting the target. The dense reward setting provides the *Tossing* and *Hitting* reward according to box position.

8.6.3.2 Locomotion

A 9-DOF bipedal planar walker is used for simulating locomotion tasks. The observation consists of the position and velocity of the torso, joint angles, and angular velocities. The action space is torque control on the 6 joints.

Reward Design Different locomotion tasks share many components of reward design, such as velocity, stability, and posture. We use the same form of reward functions, but with different hyperparameters for each task. The basic form of the reward function is as following:

$$R(s) = \lambda_{vel} \cdot \text{abs}(v_x - v_{target}) + \lambda_{alive} - \lambda_{height} \cdot \text{abs}(1.1 - \min(1.1, \Delta h)) + \\ \lambda_{angle} \cdot \cos(angle) - \lambda_{foot}(v_{right_foot} + v_{left_foot}),$$

where v_x , v_{right_foot} , and v_{left_foot} are forward velocity, right foot angular velocity, left foot angular velocity; and Δh and $angle$ are the distance between the foot and torso and the angle of the torso, respectively. The foot velocities help the agent to move its feet naturally. Δh and $angle$ are used to maintain height of the torso and encourage an upright pose.

Forward: The *Forward* task requires the walker agent to walk forward for 20 meters. To make the agent robust, we apply a random force with arbitrary magnitude and direction to a randomly selected joint every 10 timesteps.

$$\lambda_{vel} = 2, \lambda_{alive} = 1, \lambda_{height} = 2, \lambda_{angle} = 0.1, \lambda_{foot} = 0.01, \text{ and } v_{target} = 3$$

Backward: Similar to *Forward*, the *Backward* task requires the walker to walk backward for 20 meters under random forces.

$$\lambda_{vel} = 2, \lambda_{alive} = 1, \lambda_{height} = 2, \lambda_{angle} = 0.1, \lambda_{foot} = 0.01, \text{ and } v_{target} = -3$$

Balancing: In the *Balancing* task, the agent learns to balance under strong random forces for 1000 timesteps. Similar to other tasks, the random forces are applied to a random joint every 10 timesteps, but with magnitude 5 times larger.

$$\lambda_{vel} = 1, \lambda_{alive} = 1, \lambda_{height} = 0.5, \lambda_{angle} = 0.1, \lambda_{foot} = 0, \text{ and } v_{target} = 0$$

Crawling: In the *Crawling* task, a ceiling of height 1.0 and length 16 is located in front of the agent, and the agent is required to crawl under the ceiling without touching it. If the agent touches the ceiling, we terminate the episode. The task can be completed when the agent passes a point 1.5 after the ceiling and the agent gets 100 additional reward.

$$\lambda_{vel} = 2, \lambda_{alive} = 1, \lambda_{height} = 0, \lambda_{angle} = 0.1, \lambda_{foot} = 0.01, \text{ and } v_{target} = 3$$

Jumping: In the *Jumping* task, a curb of height 0.4 and length 0.2 is located in front of the walker agent. The observation contains a distance to the curb in addition to the 17-dimensional joint information, where the distance is clipped by 3. The x location of the curb is randomly chosen from [2.5, 5.5]. In addition to the reward function above, it also gets an additional 100 reward for passing the curb and $200 \cdot v_y$ when

the agent passes the front, middle, and end slices of the curb, where v_y is y-velocity. If the agent touches the curb, the agent gets -10 penalty and the episode is terminated.

$$\lambda_{vel} = 2, \lambda_{alive} = 1, \lambda_{height} = 2, \lambda_{angle} = 0.1, \lambda_{foot} = 0.01, \text{ and } v_{target} = 3$$

Patrol: The *Patrol* task is repetitive running forward and backward between two goals at $x = -2$ and $x = 2$. Once the agent touches a goal, the target is changed to another goal and the sparse reward +1 is given. The dense reward alternates between the reward functions of *Forward* and *Backward*. The agent gets the reward of *Forward* when the agent is heading toward $x = 2$ and gets the reward of *Backward*, otherwise.

Hurdle: The *Hurdle* environment consists of 5 curbs positioned at $x = \{8, 18, 28, 38, 48\}$ and requires repetitive walking and jumping behaviors. The position of each curb is randomized with a uniformly sampled value from $[-0.5, 0.5]$. The sparse reward +1 is given when the agent jumps over a curb (*i.e.* pass a point 1.5 after a curb).

The dense reward for *Hurdle* is same with *Jumping* and has 8 reward components to guide the agent to learn the desired behavior. By extensively designing dense rewards, it is possible to solve complex tasks. In comparison, our proposed method learns from sparse reward by re-using prior knowledge and doesn't require reward shaping.

Obstacle Course: The *Obstacle Course* environment replaces two curbs in *Hurdle* with a ceiling of height 1.0 and length 3. The sparse reward +1 is given when the agent jumps over a curb or passes through a ceiling (*i.e.* pass a point 1.5 after a curb or a ceiling). The dense reward is alternating between *Jumping* before the curb and *Crawling* before the ceiling.

Termination Signal Locomotion tasks except *Crawling* fail if $h < 0.8$ and *Crawling* fails if $h < 0.3$. *Forward* and *Backward* tasks are considered as success when the walker reaches to the target or 5 in front

of obstacles. *Balancing* task is considered successful when the agent does not fail for 50 timesteps. The agent succeeds on *Jumping* and *Crawling* if the agent passes the obstacles by a distance of 1.5.

Part V

Conclusion

Chapter 9

Conclusion

9.1 Summary

This dissertation describes a robot learning framework that is designed to allow robots to interpret and acquire complex skills. The key insight is to represent a skill or a task-solving procedure using programs that are structured in a formal domain-specific language (DSL). Specifically, Part II describes techniques that can infer programs from expert's demonstrations (Chapter 2) and reward functions (Chapter 3). Then, Part III introduces two lines of work that aim to efficiently acquire a set of primitive skills: meta-reinforcement learning (Chapter 4 and Chapter 5) and learning from demonstrations (Chapter 6). Finally, Part IV presents how robots can learn to execute inferred programs (Chapter 7) and hierarchically compose a set of acquired primitive skills (Chapter 8). The novelty, feasibility, and potential impact of this proposed framework have been justified by a series of publications presented at top-tier computer science and machine learning conferences.

9.2 Future Directions

To further improve the proposed robot learning framework, I plan to continue researching in the following directions.

9.2.1 Program Inference

This dissertation introduces methods that are designed to infer programs for mimicking expert's behaviors and for addressing reinforcement learning tasks. To develop more practical program inference frameworks, I plan to conduct research in the following directions.

Synthesizing programs from real-world videos. I aim to further leverage my experience in computer vision [121, 284, 285] to devise a more general program synthesis framework for synthesizing programs from more complex task specifications. This includes directions such as incorporating the scene understanding ability of computer vision models trained on large-scale datasets [58, 109, 147, 214], exploiting multimodal demonstrations (e.g. cooking instructional videos) that contain audio, captions or subtitles, manuals, etc.

Leveraging language models for program synthesis. Language models trained on large-scale datasets have achieved tremendous success in a wide range of natural language processing tasks [30, 62, 153, 338]. I believe developing and leveraging language models can significantly increase the scalability of program synthesis methods. Recently, encouraging results have been shown in leveraging language models for program synthesis in competitive programming [18, 44, 167]. In contrast, I aim to develop and leverage language models for programs that describe behaviors for agent learning.

Learning programmatic priors. The program inference works described in this dissertation rely on learning from randomly generated programs. Yet, it would be more effective to learn from goal-oriented behavioral programs, which allows for extracting meaningful programmatic behavioral priors.

Developing differentiable program executors. Existing program synthesis methods assume the availability of program executors that can execute program to produce execution results. In most cases, such a execution process is not differentiable, and therefore cannot be utilized as direct supervision (*i.e.* gradients) for learning. Developing differentiable program executors would allow learning models to be optimized based on execution results, which yield more accurate and direct supervision.

Building programmatic multi-agent learning systems. This dissertation shows encouraging results in single agent environment. I believe the advantage of the proposed framework, such as interpretability and generalization ability, can apply to multi-agent robot learning systems.

Programming from negative examples. Existing programming from examples methods [35, 46, 47, 63, 273, 286] are designed to learn from "positive" examples that demonstration inputs and outputs based on correct program execution. It would be interesting to explore how "negative" examples, that contain incorrect program execution results, can play a role in learning program inference.

9.2.2 Primitive Skill Acquisition

To efficiently acquire a set of primitive skills, this dissertation introduces meta-learning and learning from demonstration methods. I plan to continue research in meta-learning and directions that exploit the relationship among skills to further improve the learning efficiency.

9.2.3 Task Execution

The techniques proposed in this dissertation are mainly evaluated in simulated environments. Yet, to justify and improve the effectiveness of applying these techniques to real robots, I plan to work on the following directions.

Deploying proposed algorithms to real robots. I aim to investigate the advantage and the limitations of applying the proposed framework to real robot learning systems. Specifically, I plan to explore robot manipulation tasks such as object rearrangement and furniture assembly.

Researching sim-to-real techniques. I plan to research sim-to-real techniques [41, 126, 189, 227] that can translate the success achieved in simulation to real-world.

Bibliography

- [1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. “TensorFlow: A System for Large-Scale Machine Learning”. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2016.
- [2] Pieter Abbeel and Andrew Y Ng. “Apprenticeship learning via inverse reinforcement learning”. In: *International Conference on Machine Learning*. 2004.
- [3] Daniel A Abolafia, Mohammad Norouzi, Jonathan Shen, Rui Zhao, and Quoc V Le. “Neural program synthesis with priority queue training”. In: *arXiv preprint arXiv:1801.03526* (2018).
- [4] Anurag Ajay, Aviral Kumar, Pulkit Agrawal, Sergey Levine, and Ofir Nachum. “OPAL: Offline Primitive Discovery for Accelerating Offline Reinforcement Learning”. In: *International Conference on Learning Representations*. 2021.
- [5] Ferran Alet, Javier Lopez-Contreras, James Koppel, Maxwell Nye, Armando Solar-Lezama, Tomas Lozano-Perez, Leslie Kaelbling, and Joshua Tenenbaum. “A large-scale benchmark for few-shot program induction and synthesis”. In: *International Conference on Machine Learning*. 2021.
- [6] Amjad Almahairi, Sai Rajeswar, Alessandro Sordoni, Philip Bachman, and Aaron Courville. “Augmented cyclegan: Learning many-to-many mappings from unpaired data”. In: *International Conference on Machine Learning*. 2018.
- [7] Uri Alon, Omer Levy, and Eran Yahav. “code2seq: Generating sequences from structured representations of code”. In: *International Conference on Learning Representations*. 2019.
- [8] David Andre and Stuart J Russell. “Programmable reinforcement learning agents”. In: *Neural Information Processing Systems*. 2001.
- [9] David Andre and Stuart J Russell. “State abstraction for programmable reinforcement learning agents”. In: *National Conference on Artificial Intelligence*. 2002.
- [10] Jacob Andreas, Dan Klein, and Sergey Levine. “Learning with latent language”. In: *North American Chapter of the Association for Computational Linguistics*. 2017.

- [11] Jacob Andreas, Dan Klein, and Sergey Levine. “Modular multitask reinforcement learning with policy sketches”. In: *International Conference on Machine Learning*. 2017.
- [12] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. “Neural module networks”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2016.
- [13] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando de Freitas. “Learning to learn by gradient descent by gradient descent”. In: *Neural Information Processing Systems*. 2016.
- [14] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. “Hindsight experience replay”. In: *Neural Information Processing Systems*. 2017.
- [15] OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. “Learning Dexterous In-Hand Manipulation”. In: *The International Journal of Robotics Research* (2020).
- [16] Daniel Angelov, Yordan Hristov, Michael Burke, and Subramanian Ramamoorthy. “Composing Diverse Policies for Temporally Extended Tasks”. In: *IEEE Robotics and Automation Letters* (2020).
- [17] Anil Aswani, Humberto Gonzalez, S Shankar Sastry, and Claire Tomlin. “Provably safe and robust learning-based model predictive control”. In: *Automatica* (2013).
- [18] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. “Program Synthesis with Large Language Models”. In: *arXiv preprint arXiv:2108.07732* (2021).
- [19] Yusuf Aytar, Tobias Pfaff, David Budden, Thomas Paine, Ziyu Wang, and Nando de Freitas. “Playing hard exploration games by watching YouTube”. In: *Neural Information Processing Systems*. 2018.
- [20] Pierre-Luc Bacon, Jean Harb, and Doina Precup. “The Option-Critic Architecture.” In: *AAAI Conference on Artificial Intelligence*. 2017.
- [21] Dzmitry Bahdanau, Felix Hill, Jan Leike, Edward Hughes, Pushmeet Kohli, and Edward Grefenstette. “Learning to Understand Goal Specifications by Modelling Reward”. In: *International Conference on Learning Representations*. 2019.
- [22] Mihalj Bakator and Dragica Radosav. “Deep learning and medical diagnosis: A review of literature”. In: *Multimodal Technologies and Interaction* (2018).
- [23] Bram Bakker, Jürgen Schmidhuber, et al. “Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization”. In: *Intelligent Autonomous Systems*. 2004.
- [24] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. “Deepcoder: Learning to write programs”. In: *International Conference on Learning Representations*. 2017.

- [25] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. “Verifiable reinforcement learning via policy extraction”. In: *Neural Information Processing Systems*. 2018.
- [26] Harold Bekkering, Andreas Wohlschlager, and Merideth Gattis. “Imitation of gestures in children is goal-directed”. In: *The Quarterly Journal of Experimental Psychology: Section A* (2000).
- [27] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. “Unifying count-based exploration and intrinsic motivation”. In: *Neural Information Processing Systems*. 2016.
- [28] Samy Bengio, Yoshua Bengio, Jocelyn Cloutier, and Jan Gecsei. *On the Optimization of a Synaptic Learning Rule*. 1997.
- [29] Felix Berkenkamp, Matteo Turchetta, Angela P. Schoellig, and Andreas Krause. “Safe Model-Based Reinforcement Learning with Stability Guarantees”. In: *Neural Information Processing Systems*. 2017.
- [30] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. “On the opportunities and risks of foundation models”. In: *arXiv preprint arXiv:2108.07258* (2021).
- [31] Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. “Programming with a Differentiable Forth Interpreter”. In: *International Conference on Machine Learning*. 2017.
- [32] Satchuthananthavale RK Branavan, Harr Chen, Luke S Zettlemoyer, and Regina Barzilay. “Reinforcement learning for mapping instructions to actions”. In: *Assosiation of Computational Linguistics*. 2009.
- [33] SRK Branavan, Nate Kushman, Tao Lei, and Regina Barzilay. “Learning high-level planning from text”. In: *Assosiation of Computational Linguistics*. 2012.
- [34] Daniel S. Brown, Wonjoon Goo, Prabhat Nagarajan, and Scott Niekum. “Extrapolating beyond suboptimal demonstrations via inverse reinforcement learning from observations”. In: *International Conference on Machine Learning*. 2019.
- [35] Rudy R Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. “Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis”. In: *International Conference on Learning Representations*. 2018.
- [36] Michael Burke, Katie Lu, Daniel Angelov, Artūras Straižys, Craig Innes, Kartic Subr, and Subramanian Ramamoorthy. “Learning robotic ultrasound scanning using probabilistic temporal ranking”. In: *arXiv preprint arXiv:2002.01240* (2020).
- [37] Michael Burke, Svetlin Penkov, and Subramanian Ramamoorthy. “From explanation to synthesis: Compositional program induction for learning from demonstration”. In: *arXiv preprint arXiv:1902.10657* (2019).
- [38] Jonathon Cai, Richard Shin, and Dawn Song. “Making neural programming architectures generalize via recursion”. In: *International Conference on Learning Representations*. 2017.

- [39] Alexandre Campeau-Lecours, Hugo Lamontagne, Simon Latour, Philippe Fauteux, Véronique Maheu, François Boucher, Charles Deguire, and Louis-Joseph Caron L’Ecuyer. “Kinova modular robot arms for service robotics applications”. In: *Rapid Automation: Concepts, Methodologies, Tools, and Applications*. 2019.
- [40] Chia-Jung Chang, Wei Guo, Jie Zhang, Jon Newman, Shao-Hua Sun, and Matt Wilson. “Behavioral clusters revealed by end-to-end decoding from microendoscopic imaging”. In: *bioRxiv* (2021).
- [41] Yevgen Chebotar, Ankur Handa, Viktor Makoviychuk, Miles Macklin, Jan Issac, Nathan Ratliff, and Dieter Fox. “Closing the sim-to-real loop: Adapting simulation randomization with real world experience”. In: *IEEE International Conference on Robotics and Automation*. 2019.
- [42] Yevgen Chebotar, Karol Hausman, Yao Lu, Ted Xiao, Dmitry Kalashnikov, Jake Varley, Alex Irpan, Benjamin Eysenbach, Ryan Julian, Chelsea Finn, et al. “Actionable Models: Unsupervised Offline Reinforcement Learning of Robotic Skills”. In: *arXiv preprint arXiv:2104.07749* (2021).
- [43] Liushan Chen, Yu Pei, and Carlo A Furia. “Contract-based program repair without the contracts”. In: *International Conference on Automated Software Engineering*. 2017.
- [44] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. “Evaluating large language models trained on code”. In: *arXiv preprint arXiv:2107.03374* (2021).
- [45] Wei-Yu Chen, Yen-Cheng Liu, Zsolt Kira, Yu-Chiang Frank Wang, and Jia-Bin Huang. “A Closer Look at Few-shot Classification”. In: *International Conference on Learning Representations*. 2019.
- [46] Xinyun Chen, Chang Liu, and Dawn Song. “Execution-Guided Neural Program Synthesis”. In: *International Conference on Learning Representations*. 2019.
- [47] Xinyun Chen, Petros Maniatis, Rishabh Singh, Charles Sutton, Hanjun Dai, Max Lin, and Denny Zhou. “SpreadsheetCoder: Formula Prediction from Semi-structured Context”. In: *International Conference on Machine Learning*. 2021.
- [48] Xinyun Chen, Dawn Song, and Yuandong Tian. “Latent Execution for Neural Program Synthesis Beyond Domain-Specific Languages”. In: *arXiv preprint arXiv:2107.00101* (2021).
- [49] Yun-Chun Chen, Chao-Te Chou, and Yu-Chiang Frank Wang. “Learning to Learn in a Semi-supervised Fashion”. In: *European Conference on Computer Vision*. 2020.
- [50] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. *Minimalistic Gridworld Environment for OpenAI Gym*. <https://github.com/maximecb/gym-minigrid>. 2018.
- [51] Hao-Tien Lewis Chiang, Jasmine Hsu, Marek Fiser, Lydia Tapia, and Aleksandra Faust. “RL-RRT: Kinodynamic motion planning via learning reachability estimators from RL policies”. In: *IEEE Robotics and Automation Letters* (2019).
- [52] Dongkyu Choi and Pat Langley. “Learning teleoreactive logic programs from problem solving”. In: *International Conference on Inductive Logic Programming*. 2005.

- [53] Ignasi Clavera, Anusha Nagabandi, Simin Liu, Ronald S. Fearing, Pieter Abbeel, Sergey Levine, and Chelsea Finn. “Learning to Adapt in Dynamic, Real-World Environments through Meta-Reinforcement Learning”. In: *International Conference on Learning Representations*. 2019.
- [54] Ignasi Clavera, Jonas Rothfuss, John Schulman, Yasuhiro Fujita, Tamim Asfour, and Pieter Abbeel. “Model-Based Reinforcement Learning via Meta-Policy Optimization”. In: *Conference on Robot Learning*. 2018.
- [55] Raphaël Dang-Nhu. “PLANS: Neuro-Symbolic Program Learning from Videos”. In: *Neural Information Processing Systems*. 2020.
- [56] Christian Daniel, Herke Van Hoof, Jan Peters, and Gerhard Neumann. “Probabilistic inference for determining options in reinforcement learning”. In: *Machine Learning* (2016).
- [57] Sudeep Dasari, Frederik Ebert, Stephen Tian, Suraj Nair, Bernadette Bucher, Karl Schmeckpeper, Siddharth Singh, Sergey Levine, and Chelsea Finn. “RoboNet: Large-scale multi-robot learning”. In: *Conference on Robot Learning*. 2019.
- [58] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. “Imagenet: A large-scale hierarchical image database”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2009.
- [59] Misha Denil, Sergio Gómez Colmenarejo, Serkan Cabi, David Saxton, and Nando de Freitas. “Programmable agents”. In: *arXiv preprint arXiv:1706.06383* (2017).
- [60] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Subhajit Roy, et al. “Program synthesis using natural language”. In: *International Conference on Software Engineering*. 2016.
- [61] Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. “Neural Program Meta-Induction”. In: *Neural Information Processing Systems*. 2017.
- [62] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *North American Chapter of the Association for Computational Linguistics*. 2018.
- [63] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. “Robustfill: Neural program learning under noisy I/O”. In: *International Conference on Machine Learning*. 2017.
- [64] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. *OpenAI Baselines*. 2017.
- [65] Bhuwan Dhingra, Hanxiao Liu, Zhilin Yang, William W Cohen, and Ruslan Salakhutdinov. “Gated-attention readers for text comprehension”. In: *Assosiation of Computational Linguistics*. 2017.
- [66] Nat Dilokthanakul, Christos Kapliris, Nick Pawlowski, and Murray Shanahan. “Feature Control as Intrinsic Motivation for Hierarchical Reinforcement Learning”. In: *arXiv preprint arXiv:1705.06769* (2017).

- [67] Yiming Ding, Carlos Florensa, Mariano Phielipp, and Pieter Abbeel. “Goal-conditioned imitation learning”. In: *Neural Information Processing Systems*. 2019.
- [68] Ron Dorfman, Idan Shenfeld, and Aviv Tamar. “Offline Meta Learning of Exploration”. In: *Neural Information Processing Systems*. 2021.
- [69] Yan Duan, Marcin Andrychowicz, Bradly Stadie, OpenAI Jonathan Ho, Jonas Schneider, Ilya Sutskever, Pieter Abbeel, and Wojciech Zaremba. “One-shot imitation learning”. In: *Neural Information Processing Systems*. 2017.
- [70] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. “RL²: Fast Reinforcement Learning via Slow Reinforcement Learning”. In: *arXiv preprint arXiv:1611.02779* (2016).
- [71] Vincent Dumoulin, Jonathon Shlens, and Manjunath Kudlur. “A learned representation for artistic style”. In: *International Conference on Learning Representations*. 2017.
- [72] Thomas Durieux and Martin Monperrus. “Dynamoth: dynamic code synthesis for automatic program repair”. In: *International Workshop on Automation of Software Test*. 2016.
- [73] Ashley D. Edwards and Charles L. Isbell. “Perceptual Values from Observation”. In: *arXiv preprint arXiv:1905.07861* (2019).
- [74] Ashley D. Edwards, Charles L. Isbell, and Atsuo Takanishi. “Perceptual reward functions”. In: *arXiv preprint arXiv:1608.03824* (2016).
- [75] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. “Write, execute, assess: Program synthesis with a repl”. In: *Neural Information Processing Systems*. 2019.
- [76] 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”. In: *arXiv preprint arXiv:2006.08381* (2020).
- [77] Chelsea Finn, Pieter Abbeel, and Sergey Levine. “Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks”. In: *International Conference on Machine Learning*. 2017.
- [78] Chelsea Finn, Paul Christiano, Pieter Abbeel, and Sergey Levine. “A connection between generative adversarial networks, inverse reinforcement learning, and energy-based models”. In: *Adversarial Training Workshop on Neural Information Processing Systems* (2016).
- [79] Chelsea Finn and Sergey Levine. “Meta-Learning and Universality: Deep Representations and Gradient Descent can Approximate any Learning Algorithm”. In: *International Conference on Learning Representations*. 2018.
- [80] Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. “Deep spatial autoencoders for visuomotor learning”. In: *IEEE International Conference on Robotics and Automation*. 2016.

- [81] Chelsea Finn, Kelvin Xu, and Sergey Levine. “Probabilistic Model-Agnostic Meta-Learning”. In: *Neural Information Processing Systems*. 2018.
- [82] Chelsea Finn, Tianhe Yu, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. “One-shot visual imitation learning via meta-learning”. In: *Conference on Robot Learning*. 2017.
- [83] Jaime F Fisac, Anayo K Akametalu, Melanie N Zeilinger, Shahab Kaynama, Jeremy Gillula, and Claire J Tomlin. “A general safety framework for learning-based control in uncertain robotic systems”. In: *IEEE Transactions on Automatic Control* (2018).
- [84] Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. “Meta Learning Shared Hierarchies”. In: *International Conference on Learning Representations*. 2018.
- [85] Daniel Fried, Jacob Andreas, and Dan Klein. “Unified Pragmatic Models for Generating and Following Instructions”. In: *North American Chapter of the Association for Computational Linguistics*. 2017.
- [86] Daniel Fried, Ronghang Hu, Volkan Cirik, Anna Rohrbach, Jacob Andreas, Louis-Philippe Morency, Taylor Berg-Kirkpatrick, Kate Saenko, Dan Klein, and Trevor Darrell. “Speaker-Follower Models for Vision-and-Language Navigation”. In: *Neural Information Processing Systems*. 2018.
- [87] Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. “D4rl: Datasets for deep data-driven reinforcement learning”. In: *arXiv preprint arXiv:2004.07219* (2020).
- [88] Justin Fu, Katie Luo, and Sergey Levine. “Learning Robust Rewards with Adversarial Inverse Reinforcement Learning”. In: *International Conference on Learning Representations*. 2018.
- [89] Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. “Differentiable Programs with Neural Libraries”. In: *International Conference on Machine Learning*. 2017.
- [90] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.
- [91] Dibya Ghosh, Avi Singh, Aravind Rajeswaran, Vikash Kumar, and Sergey Levine. “Divide and Conquer Reinforcement Learning”. In: *International Conference on Learning Representations*. 2018.
- [92] Biraja Ghoshal and Allan Tucker. “Estimating uncertainty and interpretability in deep learning for coronavirus (COVID-19) detection”. In: *arXiv preprint arXiv:2003.10769* (2020).
- [93] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. “Automated program repair”. In: *Communications of the ACM* (2019).
- [94] Erin Grant, Chelsea Finn, Sergey Levine, Trevor Darrell, and Thomas Griffiths. “Recasting Gradient-Based Meta-Learning as Hierarchical Bayes”. In: *International Conference on Learning Representations*. 2018.
- [95] Alex Graves, Greg Wayne, and Ivo Danihelka. “Neural turing machines”. In: *arXiv preprint arXiv:1410.5401* (2014).

- [96] 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”. In: *Nature* (2016).
- [97] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. “Learning to transduce with unbounded memory”. In: *Neural Information Processing Systems*. 2015.
- [98] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. “Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates”. In: *IEEE International Conference on Robotics and Automation*. 2017.
- [99] Aditya Gudimella, Ross Story, Matineh Shaker, Ruofan Kong, Matthew Brown, Victor Shnayder, and Marcos Campos. “Deep Reinforcement Learning for Dexterous Manipulation with Concept Networks”. In: *arXiv preprint arXiv: 1709.06977* (2017).
- [100] Caglar Gulcehre, Ziyu Wang, Alexander Novikov, Tom Le Paine, Sergio Gómez Colmenarejo, Konrad Zolna, Rishabh Agarwal, Josh Merel, Daniel Mankowitz, Cosmin Paduraru, et al. “RL Unplugged: Benchmarks for Offline Reinforcement Learning”. In: *arXiv preprint arXiv:2006.13888* (2020).
- [101] Abhishek Gupta, Vikash Kumar, Corey Lynch, Sergey Levine, and Karol Hausman. “Relay Policy Learning: Solving Long-Horizon Tasks via Imitation and Reinforcement Learning”. In: *Conference on Robot Learning*. 2019.
- [102] Abhishek Gupta, Russell Mendonca, YuXuan Liu, Pieter Abbeel, and Sergey Levine. “Meta-Reinforcement Learning of Structured Exploration Strategies”. In: *Neural Information Processing Systems*. 2018.
- [103] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. “Deepfix: Fixing common c language errors by deep learning”. In: *AAAI Conference on Artificial Intelligence*. 2017.
- [104] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor”. In: *International Conference on Machine Learning*. 2018.
- [105] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. “Dream to Control: Learning Behaviors by Latent Imagination”. In: *International Conference on Learning Representations*. 2020.
- [106] A. Hakobyan, G. C. Kim, and I. Yang. “Risk-Aware Motion Planning and Control Using CVaR-Constrained Optimization”. In: *IEEE Robotics and Automation Letters* (2019).
- [107] Chi Han, Jiayuan Mao, Chuang Gan, Josh Tenenbaum, and Jiajun Wu. “Visual Concept-Metaconcept Learning”. In: *Neural Information Processing Systems*. 2019.
- [108] Karol Hausman, Jost Tobias Springenberg, Ziyu Wang, Nicolas Heess, and Martin Riedmiller. “Learning an Embedding Space for Transferable Robot Skills”. In: *International Conference on Learning Representations*. 2018.

- [109] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. “Mask r-cnn”. In: *International Conference on Computer Vision*. 2017.
- [110] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. “Deep residual learning for image recognition”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2016.
- [111] Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, S. M. Ali Eslami, Martin A. Riedmiller, and David Silver. “Emergence of Locomotion Behaviours in Rich Environments”. In: *arXiv preprint arXiv: 1707.02286* (2017).
- [112] Lukas Hewing, Juraj Kabzan, and Melanie N. Zeilinger. “Cautious Model Predictive Control Using Gaussian Process Regression”. In: *IEEE Transactions on Control System Technology* (2019).
- [113] Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. “beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework”. In: *International Conference on Learning Representations*. 2017.
- [114] Jonathan Ho and Stefano Ermon. “Generative adversarial imitation learning”. In: *Neural Information Processing Systems*. 2016.
- [115] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural Computation* (1997).
- [116] Nicola J Hodges, A Mark Williams, Spencer J Hayes, and Gavin Breslin. “What is modelled during observational learning?” In: *Journal of sports sciences* (2007).
- [117] Joey Hong, David Dohan, Rishabh Singh, Charles Sutton, and Manzil Zaheer. “Latent Programmer: Discrete Latent Codes for Program Synthesis”. In: *International Conference on Machine Learning*. 2021.
- [118] Hexiang Hu, Liyu Chen, Boqing Gong, and Fei Sha. “Synthesized Policies for Transfer and Adaptation across Tasks and Environments”. In: *Neural Information Processing Systems*. 2018.
- [119] Jie Hu, Li Shen, and Gang Sun. “Squeeze-and-excitation networks”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018.
- [120] 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: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019.
- [121] Minyoung Huh, Shao-Hua Sun, and Ning Zhang. “Feedback Adversarial Learning: Spatial Feedback for Improving Generative Adversarial Networks”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019.
- [122] Jan Humplík, Alexandre Galashov, Leonard Hasenclever, Pedro A Ortega, Yee Whye Teh, and Nicolas Heess. “Meta reinforcement learning as task inference”. In: *arXiv preprint arXiv:1905.06424* (2019).

- [123] Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. “Synthesizing Programmatic Policies that Inductively Generalize”. In: *International Conference on Learning Representations*. 2020.
- [124] Ayush Jain, Andrew Szot, and Joseph J. Lim. “Generalization to new actions in reinforcement learning”. In: *International Conference on Machine Learning*. 2020.
- [125] Stephen James, Andrew J Davison, and Edward Johns. “Transferring End-to-End Visuomotor Control from Simulation to Real World for a Multi-Stage Task”. In: *Conference on Robot Learning*. 2017.
- [126] Stephen James, Paul Wohlhart, Mrinal Kalakrishnan, Dmitry Kalashnikov, Alex Irpan, Julian Ibarz, Sergey Levine, Raia Hadsell, and Konstantinos Bousmalis. “Sim-to-real via sim-to-sim: Data-efficient robotic grasping via randomized-to-canonical adaptation networks”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019.
- [127] Michael Janner, Karthik Narasimhan, and Regina Barzilay. “Representation learning for grounded spatial reasoning”. In: *Assosiation of Computational Linguistics*. 2018.
- [128] Jermsak Jermsurawong and Nizar Habash. “Predicting the Structure of Cooking Recipes”. In: *Empirical Methods in Natural Language Processing*. 2015.
- [129] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. “Shaping program repair space with existing patches and similar code”. In: *ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2018.
- [130] I.T. Jolliffe. *Principal Component Analysis*. Springer Verlag, 1986.
- [131] Armand Joulin and Tomas Mikolov. “Inferring algorithmic patterns with stack-augmented recurrent nets”. In: *Neural Information Processing Systems*. 2015.
- [132] Łukasz Kaiser and Ilya Sutskever. “Neural gpus learn algorithms”. In: *International Conference on Learning Representations*. 2016.
- [133] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, and Sergey Levine. “Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation”. In: *Conference on Robot Learning*. 2018.
- [134] Dmitry Kalashnikov, Jacob Varley, Yevgen Chebotar, Benjamin Swanson, Rico Jonschkowski, Chelsea Finn, Sergey Levine, and Karol Hausman. “MT-Opt: Continuous Multi-Task Robotic Reinforcement Learning at Scale”. In: *arXiv preprint arXiv:2104.08212* (2021).
- [135] Russell Kaplan, Christopher Sauer, and Alexander Sosa. “Beating Atari with Natural Language Guided Reinforcement Learning”. In: *arXiv preprint arXiv:1704.05539* (2017).
- [136] Tero Karras, Samuli Laine, and Timo Aila. “A style-based generator architecture for generative adversarial networks”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019.

- [137] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. “Vizdoom: A doom-based ai research platform for visual reinforcement learning”. In: *Computational Intelligence and Games*. 2016.
- [138] Chloé Kiddon, Ganesa Thandavam Ponnuraj, Luke Zettlemoyer, and Yejin Choi. “Mise en Place: Unsupervised Interpretation of Instructional Recipes”. In: *Empirical Methods in Natural Language Processing*. 2015.
- [139] Taesup Kim, Jaesik Yoon, Ousmane Dia, Sungwoong Kim, Yoshua Bengio, and Sungjin Ahn. “Bayesian Model-Agnostic Meta-Learning”. In: *Neural Information Processing Systems*. 2018.
- [140] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *International Conference on Learning Representations*. 2015.
- [141] Diederik P Kingma and Max Welling. “Auto-Encoding Variational Bayes”. In: *International Conference on Learning Representations*. 2014.
- [142] Jens Kober, Katharina Mülling, Oliver Krömer, Christoph H Lampert, Bernhard Schölkopf, and Jan Peters. “Movement templates for learning of hitting and batting”. In: *IEEE International Conference on Robotics and Automation*. 2010.
- [143] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. “Siamese neural networks for one-shot image recognition”. In: *Deep Learning Workshop at International Conference on Machine Learning*. 2015.
- [144] George Konidaris, Leslie Pack Kaelbling, and Tomas Lozano-Perez. “From skills to symbols: Learning symbolic representations for abstract high-level planning”. In: *Journal of Artificial Intelligence Research* (2018).
- [145] Ilya Kostrikov, Kumar Krishna Agrawal, Debidatta Dwibedi, Sergey Levine, and Jonathan Tompson. “Discriminator-Actor-Critic: Addressing Sample Inefficiency and Reward Bias in Adversarial Imitation Learning”. In: *International Conference on Learning Representations*. 2019.
- [146] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. “Fixminer: Mining relevant fix patterns for automated program repair”. In: *Empirical Software Engineering* (2020).
- [147] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Neural Information Processing Systems*. 2012.
- [148] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. “Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation”. In: *Neural Information Processing Systems*. 2016.
- [149] Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. “Rma: Rapid motor adaptation for legged robots”. In: *Robotics: Science and Systems*. 2021.
- [150] Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. “Conservative q-learning for offline reinforcement learning”. In: *Neural Information Processing Systems*. 2020.

- [151] Brenden Lake, Ruslan Salakhutdinov, Jason Gross, and Joshua Tenenbaum. “One shot learning of simple visual concepts”. In: *Conference of the Cognitive Science Society*. 2011.
- [152] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. “Simple and scalable predictive uncertainty estimation using deep ensembles”. In: *Neural Information Processing Systems*. 2017.
- [153] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. “Albert: A lite bert for self-supervised learning of language representations”. In: *arXiv preprint arXiv:1909.11942* (2019).
- [154] Mikel Landajuela, Brenden K Petersen, Sookyoung Kim, Claudio P Santiago, Ruben Glatt, Nathan Mundhenk, Jacob F Pettit, and Daniel Faissol. “Discovering symbolic policies with deep reinforcement learning”. In: *International Conference on Machine Learning*. 2021.
- [155] Miguel Lázaro-Gredilla, Dianhuan Lin, J Swaroop Guntupalli, and Dileep George. “Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs”. In: *Science Robotics* (2019).
- [156] Hoang Le, Nan Jiang, Alekh Agarwal, Miroslav Dudik, Yisong Yue, and Hal Daumé III. “Hierarchical Imitation and Reinforcement Learning”. In: *International Conference on Machine Learning*. 2018.
- [157] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. “Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks”. In: *International Conference on Machine Learning*. 2019.
- [158] Yoonho Lee and Seungjin Choi. “Gradient-Based Meta-Learning with Learned Layerwise Metric and Subspace”. In: *International Conference on Machine Learning*. 2018.
- [159] Youngwoon Lee, Edward S Hu, Zhengyu Yang, and Joseph J. Lim. “To follow or not to follow: Selective imitation learning from observations”. In: *Conference on Robot Learning*. 2019.
- [160] Youngwoon Lee, Shao-Hua Sun, Sriram Somasundaram, Edward S. Hu, and Joseph J. Lim. “Composing Complex Skills by Learning Transition Policies”. In: *International Conference on Learning Representations*. 2019.
- [161] Youngwoon Lee, Andrew Szot, Shao-Hua Sun, and Joseph J. Lim. “Generalizable Imitation Learning from Observation via Inferring Goal Proximity”. In: *Neural Information Processing Systems*. 2021.
- [162] Vladimir Iosifovich Levenshtein. “Binary codes capable of correcting deletions, insertions and reversals”. In: *Soviet Physics Doklady* (1966).
- [163] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. “Offline reinforcement learning: Tutorial, review, and perspectives on open problems”. In: *arXiv preprint arXiv:2005.01643* (2020).
- [164] Andrew Levy, Robert Platt, and Kate Saenko. “Hierarchical Actor-Critic”. In: *arXiv preprint arXiv:1712.00948* (2017).
- [165] Ke Li and Jitendra Malik. “Learning to Optimize”. In: *International Conference on Learning Representations*. 2016.

- [166] Yi Li, Shaohua Wang, and Tien N Nguyen. “DLfix: Context-based code transformation learning for automated program repair”. In: *International Conference on Software Engineering*. 2020.
- [167] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweis, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. *Competition-Level Code Generation with AlphaCode*. 2022. URL: <https://www.deeplearning.com/blog/article/Competitive-programming-with-AlphaCode>.
- [168] Yujia Li, Felix Gimeno, Pushmeet Kohli, and Oriol Vinyals. “Strong generalization and efficiency in neural programs”. In: *arXiv preprint arXiv:2007.03629* (2020).
- [169] Yuan-Hong Liao, Xavier Puig, Marko Boben, Antonio Torralba, and Sanja Fidler. “Synthesizing Environment-Aware Activities via Activity Sketches”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019.
- [170] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. “Continuous control with deep reinforcement learning”. In: *International Conference on Learning Representations*. 2016.
- [171] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. “NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system”. In: *International Conference on Language Resources and Evaluation*. 2018.
- [172] Zachary C Lipton. “The Mythos of Model Interpretability”. In: *Workshop on Human Interpretability in Machine Learning at International Conference on Machine Learning*. 2016.
- [173] Paweł Liskowski, Krzysztof Krawiec, Nihat Engin Toklu, and Jerry Swan. “Program Synthesis as Latent Continuous Optimization: Evolutionary Search in Neural Embeddings”. In: *Genetic and Evolutionary Computation Conference*. 2020.
- [174] Evan Z Liu, Aditi Raghunathan, Percy Liang, and Chelsea Finn. “Decoupling exploration and exploitation for meta-reinforcement learning without sacrifices”. In: *International Conference on Machine Learning*. 2021.
- [175] Yunchao Liu, Jiajun Wu, Zheng Wu, Daniel Ritchie, William T. Freeman, and Joshua B. Tenenbaum. “Learning to Describe Scenes with Programs”. In: *International Conference on Learning Representations*. 2019.
- [176] YuXuan Liu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. “Imitation from observation: Learning to imitate behaviors from raw video via context translation”. In: *IEEE International Conference on Robotics and Automation*. 2018.
- [177] Jonathan Long, Evan Shelhamer, and Trevor Darrell. “Fully convolutional networks for semantic segmentation”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2015.

- [178] Thang Luong, Hieu Pham, and Christopher D. Manning. “Effective Approaches to Attention-based Neural Machine Translation”. In: *Empirical Methods in Natural Language Processing*. 2015.
- [179] Corey Lynch, Mohi Khansari, Ted Xiao, Vikash Kumar, Jonathan Tompson, Sergey Levine, and Pierre Sermanet. “Learning latent plans from play”. In: *Conference on Robot Learning*. 2020.
- [180] Laurens van der Maaten and Geoffrey Hinton. “Visualizing data using t-SNE”. In: *Journal of Machine Learning Research* (2008).
- [181] Subhransu Maji, Esa Rahtu, Juho Kannala, Matthew Blaschko, and Andrea Vedaldi. “Fine-Grained Visual Classification of Aircraft”. In: *arXiv preprint arxiv:1306.5151* (2013).
- [182] Jonathan Malmaud, Earl Wagner, Nancy Chang, and Kevin Murphy. “Cooking with Semantics”. In: *Workshop on Semantic Parsing at Association for Computational Linguistics*. 2014.
- [183] Ajay Mandlekar, Yuke Zhu, Animesh Garg, Jonathan Booher, Max Spero, Albert Tung, Julian Gao, John Emmons, Anchit Gupta, Emre Orbay, Silvio Savarese, and Li Fei-Fei. “RoboTurk: A Crowdsourcing Platform for Robotic Skill Learning through Imitation”. In: *Conference on Robot Learning*. 2018.
- [184] Jiayuan Mao, Honghua Dong, and Joseph J. Lim. “Universal Agent for Disentangling Environments and Tasks”. In: *International Conference on Learning Representations*. 2018.
- [185] 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: *International Conference on Learning Representations*. 2019.
- [186] Xudong Mao, Qing Li, Haoran Xie, Raymond YK Lau, Zhen Wang, and Stephen Paul Smolley. “Least squares generative adversarial networks”. In: *International Conference on Computer Vision*. 2017.
- [187] Jarryd Martin, S Suraj Narayanan, Tom Everitt, and Marcus Hutter. “Count-based exploration in feature space for reinforcement learning”. In: *AAAI Conference on Artificial Intelligence*. 2017.
- [188] Maja J Mataric. “Reward functions for accelerated learning”. In: *International Conference on Machine Learning*. 1994.
- [189] Jan Matas, Stephen James, and Andrew J Davison. “Sim-to-real reinforcement learning for deformable object manipulation”. In: *Conference on Robot Learning*. 2018.
- [190] Josh Merel, Yuval Tassa, Dhruva TB, Sriram Srinivasan, Jay Lemmon, Ziyu Wang, Greg Wayne, and Nicolas Heess. “Learning human behaviors from motion capture by adversarial imitation”. In: *arXiv preprint arXiv: 1707.02201* (2017).
- [191] Josh Merel, Saran Tunyasuvunakool, Arun Ahuja, Yuval Tassa, Leonard Hasenclever, Vu Pham, Tom Erez, Greg Wayne, and Nicolas Heess. “Catch & Carry: Reusable Neural Controllers for Vision-Guided Whole-Body Tasks”. In: *ACM Transactions on Graphics* (2020).

- [192] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. “DeepDelta: learning to repair compilation errors”. In: *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019.
- [193] Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. “A Simple Neural Attentive Meta-Learner”. In: *International Conference on Learning Representations*. 2018.
- [194] Dipendra Misra, Andrew Bennett, Valts Blukis, Eyvind Niklasson, Max Shatkhin, and Yoav Artzi. “Mapping instructions to actions in 3D environments with visual goal prediction”. In: *Empirical Methods in Natural Language Processing*. 2018.
- [195] Dipendra Misra, John Langford, and Yoav Artzi. “Mapping instructions and visual observations to actions with reinforcement learning”. In: *Empirical Methods in Natural Language Processing*. 2017.
- [196] Eric Mitchell, Rafael Rafailov, Xue Bin Peng, Sergey Levine, and Chelsea Finn. “Offline Meta-Reinforcement Learning with Advantage Weighting”. In: *International Conference on Machine Learning*. 2021.
- [197] Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. “Spectral Normalization for Generative Adversarial Networks”. In: *International Conference on Learning Representations*. 2018.
- [198] 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: *International Conference on Machine Learning*. 2016.
- [199] Volodymyr Mnih, Nicolas Heess, Alex Graves, and koray kavukcuoglu koray. “Recurrent Models of Visual Attention”. In: *Neural Information Processing Systems*. 2014.
- [200] 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”. In: *Nature* (2015).
- [201] Katharina Mülling, Jens Kober, Oliver Kroemer, and Jan Peters. “Learning to select and generalize striking movements in robot table tennis”. In: *The International Journal of Robotics Research* (2013).
- [202] Tsendsuren Munkhdalai and Hong Yu. “Meta Networks”. In: *International Conference on Machine Learning*. 2017.
- [203] Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. “Data-efficient hierarchical reinforcement learning”. In: *Neural Information Processing Systems*. 2018.
- [204] Anusha Nagabandi, Chelsea Finn, and Sergey Levine. “Deep online learning via meta-learning: Continual adaptation for model-based rl”. In: *International Conference on Learning Representations*. 2019.
- [205] Taewook Nam, Shao-Hua Sun, Karl Pertsch, Sung Ju Hwang, and Joseph J. Lim. “Skill-based Meta-Reinforcement Learning”. In: *Meta-Learning Workshop at NeurIPS*. 2021.

- [206] Taewook Nam, Shao-Hua Sun, Karl Pertsch, Sung Ju Hwang, and Joseph J. Lim. “Skill-based Meta-Reinforcement Learning”. In: *Deep Reinforcement Learning Workshop at NeurIPS*. 2021.
- [207] Taewook Nam, Shao-Hua Sun, Karl Pertsch, Sung Ju Hwang, and Joseph J. Lim. “Skill-based Meta-Reinforcement Learning”. In: *International Conference on Learning Representations*. 2022.
- [208] Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. “Neural programmer: Inducing latent programs with gradient descent”. In: *International Conference on Learning Representations*. 2015.
- [209] Andrew Y Ng, Daishi Harada, and Stuart Russell. “Policy invariance under reward transformations: Theory and application to reward shaping”. In: *International Conference on Machine Learning*. 1999.
- [210] Andrew Y Ng, Stuart J Russell, et al. “Algorithms for inverse reinforcement learning.” In: *International Conference on Machine Learning*. 2000.
- [211] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. “Semfix: Program repair via semantic analysis”. In: *Conference on Software Engineering*. 2013.
- [212] Alex Nichol and John Schulman. “Reptile: a Scalable Metalearning Algorithm”. In: *arXiv preprint arXiv:1803.02999* (2018).
- [213] Scott Niekum, Sarah Osentoski, George Konidaris, Sachin Chitta, Bhaskara Marthi, and Andrew G Barto. “Learning grounded finite-state representations from unstructured demonstrations”. In: *The International Journal of Robotics Research* (2015).
- [214] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. “Learning deconvolution network for semantic segmentation”. In: *International Conference on Computer Vision*. 2015.
- [215] Maxwell Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B. Tenenbaum, and Armando Solar-Lezama. “Representing Partial Programs with Blended Abstract Semantics”. In: *International Conference on Learning Representations*. 2021.
- [216] Junhyuk Oh, Satinder Singh, Honglak Lee, and Pushmeet Kohli. “Zero-shot task generalization with multi-task deep reinforcement learning”. In: *International Conference on Machine Learning*. 2017.
- [217] Boris N. Oreshkin, Pau Rodriguez, and Alexandre Lacoste. “TADAM: Task dependent adaptive metric for improved few-shot learning”. In: *Neural Information Processing Systems*. 2018.
- [218] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. “Deep exploration via bootstrapped DQN”. In: *Neural Information Processing Systems*. 2016.
- [219] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. “Neuro-symbolic program synthesis”. In: *International Conference on Learning Representations*. 2017.
- [220] Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. “Semantic Image Synthesis with Spatially-Adaptive Normalization”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019.

- [221] Ronald Parr and Stuart J Russell. “Reinforcement learning with hierarchies of machines”. In: *Neural Information Processing Systems*. 1998.
- [222] Peter Pastor, Heiko Hoffmann, Tamim Asfour, and Stefan Schaal. “Learning and generalization of motor skills by learning from demonstration”. In: *IEEE International Conference on Robotics and Automation*. 2009.
- [223] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. “Automatic Differentiation in PyTorch”. In: *Autodiff Workshop at Neural Information Processing System*. 2017.
- [224] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. “Curiosity-driven exploration by self-supervised prediction”. In: *International Conference on Machine Learning*. 2017.
- [225] Deepak Pathak, Parsa Mahmoudieh, Guanghao Luo, Pulkit Agrawal, Dian Chen, Yide Shentu, Evan Shelhamer, Jitendra Malik, Alexei A Efros, and Trevor Darrell. “Zero-shot visual imitation”. In: *International Conference on Learning Representations*. 2018.
- [226] Richard E Pattis. *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, Inc., 1981.
- [227] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. “Sim-to-real transfer of robotic control with dynamics randomization”. In: *IEEE International Conference on Robotics and Automation*. 2018.
- [228] Xue Bin Peng, Glen Berseth, KangKang Yin, and Michiel Van De Panne. “Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning”. In: *ACM Transactions on Graphics* (2017).
- [229] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. “GloVe: Global Vectors for Word Representation”. In: *Empirical Methods in Natural Language Processing*. 2014.
- [230] Ethan Perez, Harm De Vries, Florian Strub, Vincent Dumoulin, and Aaron Courville. “Learning visual reasoning without strong priors”. In: (2017).
- [231] Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron Courville. “FiLM: Visual Reasoning with a General Conditioning Layer”. In: *AAAI Conference on Artificial Intelligence*. 2018.
- [232] Karl Pertsch, Youngwoon Lee, and Joseph J. Lim. “Accelerating Reinforcement Learning with Learned Skill Priors”. In: *Conference on Robot Learning*. 2020.
- [233] Karl Pertsch, Youngwoon Lee, Yue Wu, and Joseph J. Lim. “Demonstration-Guided Reinforcement Learning with Learned Skills”. In: *Conference on Robot Learning*. 2021.
- [234] Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. “Deep Contextualized Word Representations”. In: *North American Chapter of the Association for Computational Linguistics*. 2018.

- [235] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, et al. “Multi-goal reinforcement learning: Challenging robotics environments and request for research”. In: *arXiv preprint arXiv:1802.09464* (2018).
- [236] Dean A Pomerleau. “Alvinn: An autonomous land vehicle in a neural network”. In: *Neural Information Processing Systems*. 1989.
- [237] Dean A Pomerleau. “Efficient training of artificial neural networks for autonomous navigation”. In: *Neural Computation* (1991).
- [238] Vitchyr H Pong, Ashvin Nair, Laura Smith, Catherine Huang, and Sergey Levine. “Offline Meta-Reinforcement Learning with Online Self-Supervision”. In: *arXiv preprint arXiv:2107.03974* (2021).
- [239] Garima Pruthi, Frederick Liu, Satyen Kale, and Mukund Sundararajan. “Estimating Training Data Influence by Tracing Gradient Descent”. In: *Neural Information Processing Systems*. 2020.
- [240] Rafael Rafailov, Tianhe Yu, Aravind Rajeswaran, and Chelsea Finn. “Visual Adversarial Imitation Learning using Variational Models”. In: *arXiv preprint arXiv:2107.08829* (2021).
- [241] Colin Raffel, Minh-Thang Luong, Peter J. Liu, Ron J. Weiss, and Douglas Eck. “Online and Linear-Time Attention by Enforcing Monotonic Alignments”. In: *International Conference on Machine Learning*. 2017.
- [242] Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. “Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations”. In: *Robotics: Science and Systems*. 2018.
- [243] Kate Rakelly, Aurick Zhou, Chelsea Finn, Sergey Levine, and Deirdre Quillen. “Efficient off-policy meta-reinforcement learning via probabilistic context variables”. In: *International Conference on Machine Learning*. 2019.
- [244] Sachin Ravi and Hugo Larochelle. “Optimization as a Model for Few-Shot Learning”. In: *International Conference on Learning Representations*. 2017.
- [245] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. “Compositional program synthesis from natural language and examples”. In: *International Joint Conference on Artificial Intelligence*. 2015.
- [246] Siddharth Reddy, Anca D. Dragan, and Sergey Levine. “SQIL: Imitation Learning via Reinforcement Learning with Sparse Rewards”. In: *International Conference on Learning Representations*. 2020.
- [247] Scott Reed and Nando De Freitas. “Neural programmer-interpreters”. In: *International Conference on Learning Representations*. 2016.
- [248] John Co-Reyes, YuXuan Liu, Abhishek Gupta, Benjamin Eysenbach, Pieter Abbeel, and Sergey Levine. “Self-Consistent Trajectory Autoencoder: Hierarchical Reinforcement Learning with Trajectory Embeddings”. In: *International Conference on Machine Learning*. 2018.

- [249] John D Co-Reyes, Abhishek Gupta, Suvansh Sanjeev, Nick Altieri, John DeNero, Pieter Abbeel, and Sergey Levine. “Guiding policies with language via meta-learning”. In: *International Conference on Learning Representations*. 2019.
- [250] Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom van de Wiele, Vlad Mnih, Nicolas Heess, and Jost Tobias Springenberg. “Learning by Playing Solving Sparse Reward Tasks from Scratch”. In: *International Conference on Machine Learning*. 2018.
- [251] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. “A reduction of imitation learning and structured prediction to no-regret online learning”. In: *International Conference on Artificial Intelligence and Statistics*. 2011.
- [252] Jonas Rothfuss, Dennis Lee, Ignasi Clavera, Tamim Asfour, Pieter Abbeel, Dmitriy Shingarey, Lukas Kaul, Tamim Asfour, C Dometios Athanasios, You Zhou, et al. “ProMP: Proximal Meta-Policy Search”. In: *International Conference on Learning Representations*. 2019.
- [253] Reuven Y Rubinstein. “Optimization of computer simulation models with rare events”. In: *European Journal of Operational Research* (1997).
- [254] Andrei A. Rusu, Dushyant Rao, Jakub Sygnowski, Oriol Vinyals, Razvan Pascanu, Simon Osindero, and Raia Hadsell. “Meta-Learning with Latent Embedding Optimization”. In: *International Conference on Learning Representations*. 2019.
- [255] Dorsa Sadigh and Ashish Kapoor. “Safe Control under Uncertainty with Probabilistic Signal Temporal Logic”. In: *Robotics: Science and Systems*. 2016.
- [256] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. “Meta-Learning with Memory-Augmented Neural Networks”. In: *International Conference on Machine Learning*. 2016.
- [257] Adam Santoro, David Raposo, David G Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Tim Lillicrap. “A simple neural network module for relational reasoning”. In: *Neural Information Processing Systems*. 2017.
- [258] Stefan Schaal. “Learning from demonstration”. In: *Neural Information Processing Systems*. 1997.
- [259] Stefan Schaal, Jan Peters, Jun Nakanishi, and Auke Ijspeert. “Learning Movement Primitives”. In: *Robotics Research*. 2005.
- [260] Jürgen Schmidhuber. “Towards compositional learning with dynamic neural networks”. In: (1990).
- [261] Jurgen Schmidhuber. “Evolutionary principles in self-referential learning. (On learning how to learn: The meta-meta-... hook.)” Diploma Thesis. 1987.
- [262] Jürgen Schmidhuber, Jieyu Zhao, and Nicol N Schraudolph. “Reinforcement learning with self-modifying policies”. In: *Learning to learn*. 1998.
- [263] Jürgen Schmidhuber, Jieyu Zhao, and Marco Wiering. “Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement”. In: *Machine Learning* (1997).

- [264] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. “Trust region policy optimization”. In: *International Conference on Machine Learning*. 2015.
- [265] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).
- [266] Eric Schulte, Stephanie Forrest, and Westley Weimer. “Automated program repair through the evolution of assembly code”. In: *International Conference on Automated Software Engineering*. 2010.
- [267] Pierre Sermanet, Corey Lynch, Yevgen Chebotar, Jasmine Hsu, Eric Jang, Stefan Schaal, Sergey Levine, and Google Brain. “Time-contrastive networks: Self-supervised learning from video”. In: *IEEE International Conference on Robotics and Automation*. 2018.
- [268] Pierre Sermanet, Kelvin Xu, and Sergey Levine. “Unsupervised perceptual rewards for imitation learning”. In: *Robotics: Science and Systems*. 2017.
- [269] Archit Sharma, Shixiang Gu, Sergey Levine, Vikash Kumar, and Karol Hausman. “Dynamics-aware unsupervised discovery of skills”. In: *International Conference on Learning Representations*. 2020.
- [270] Dinggang Shen, Guorong Wu, and Heung-Il Suk. “Deep learning in medical image analysis”. In: *Annual review of biomedical engineering* (2017).
- [271] Owen Shen. *Interpretability in ML: A Broad Overview*. 2020. URL: <https://mlu.red/muse/52906366310>.
- [272] Nobuyuki Shimizu and Andrew Haas. “Learning to follow navigational route instructions”. In: *International Joint Conference on Artificial Intelligence*. 2009.
- [273] Eui Chul Shin, Illia Polosukhin, and Dawn Song. “Improving neural program synthesis with inferred execution traces”. In: *Neural Information Processing Systems*. 2018.
- [274] Pranav Shyam, Shubham Gupta, and Ambedkar Dukkipati. “Attentive Recurrent Comparators”. In: *International Conference on Machine Learning*. 2017.
- [275] Noah Y Siegel, Jost Tobias Springenberg, Felix Berkenkamp, Abbas Abdolmaleki, Michael Neunert, Thomas Lampe, Roland Hafner, and Martin Riedmiller. “Keep doing what worked: Behavioral modelling priors for offline reinforcement learning”. In: *International Conference on Learning Representations*. 2020.
- [276] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* (2016).
- [277] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* (2018).
- [278] Tom Silver, Kelsey R Allen, Alex K Lew, Leslie Pack Kaelbling, and Josh Tenenbaum. “Few-shot Bayesian imitation learning with logical program policies”. In: *AAAI Conference on Artificial Intelligence*. 2020.

- [279] Amitojdeep Singh, Sourya Sengupta, and Vasudevan Lakshminarayanan. “Explainable deep learning models in medical image analysis”. In: *Journal of Imaging* (2020).
- [280] Jake Snell, Kevin Swersky, and Richard Zemel. “Prototypical Networks for Few-shot Learning”. In: *Neural Information Processing Systems*. 2017.
- [281] Sungryull Sohn, Junhyuk Oh, and Honglak Lee. “Hierarchical Reinforcement Learning for Zero-shot Generalization with Subtask Dependencies”. In: *Neural Information Processing Systems*. 2018.
- [282] Aravind Srinivas, Allan Jabri, Pieter Abbeel, Sergey Levine, and Chelsea Finn. “Universal Planning Networks: Learning Generalizable Representations for Visuomotor Control”. In: *International Conference on Machine Learning*. 2018.
- [283] Bradly C. Stadie, Pieter Abbeel, and Ilya Sutskever. “Third Person Imitation Learning”. In: *International Conference on Learning Representations*. 2017.
- [284] Shao-Hua Sun, Shang-Pu Fan, and Yu-Chiang Frank Wang. “Exploiting image structural similarity for single image rain removal”. In: *IEEE International Conference on Image Processing*. 2014.
- [285] Shao-Hua Sun, Minyoung Huh, Yuan-Hong Liao, Ning Zhang, and Joseph J. Lim. “Multi-view to Novel View: Synthesizing Novel Views with Self-Learned Confidence”. In: *European Conference on Computer Vision*. 2018.
- [286] Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. “Neural program synthesis from diverse demonstration videos”. In: *International Conference on Machine Learning*. 2018.
- [287] Shao-Hua Sun, Te-Lin Wu, and Joseph J. Lim. “Program Guided Agent”. In: *International Conference on Learning Representations*. 2020.
- [288] 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: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018.
- [289] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to sequence learning with neural networks”. In: *Neural Information Processing Systems*. 2014.
- [290] Richard S Sutton, Doina Precup, and Satinder Singh. “Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning”. In: *Artificial intelligence* (1999).
- [291] Richard Stuart Sutton. “Temporal credit assignment in reinforcement learning”. PhD thesis. University of Massachusetts, Amherst, 1984.
- [292] Gokul Swamy, Sanjiban Choudhury, J Andrew Bagnell, and Steven Wu. “Of Moments and Matching: A Game-Theoretic Framework for Closing the Imitation Gap”. In: *International Conference on Machine Learning*. 2021.

- [293] Kai Sheng Tai, Richard Socher, and Christopher D Manning. “Improved semantic representations from tree-structured long short-term memory networks”. In: *Assosiation of Computational Linguistics*. 2015.
- [294] Yee Teh, Victor Bapst, Wojciech M. Czarnecki, John Quan, James Kirkpatrick, Raia Hadsell, Nicolas Heess, and Razvan Pascanu. “Distral: Robust multitask reinforcement learning”. In: *Neural Information Processing Systems*. 2017.
- [295] Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew R Walter, Ashis Gopal Banerjee, Seth J Teller, and Nicholas Roy. “Understanding Natural Language Commands for Robotic Navigation and Mobile Manipulation.” In: *AAAI Conference on Artificial Intelligence*. 2011.
- [296] Brijen Thananjeyan, Ashwin Balakrishna, Ugo Rosolia, Felix Li, Rowan McAllister, Joseph E. Gonzalez, Sergey Levine, Francesco Borrelli, and Ken Goldberg. “Safety Augmented Value Estimation From Demonstrations (SAVED): Safe Deep Model-Based RL for Sparse Cost Robotic Tasks”. In: *IEEE Robotics and Automation Letters* (2020).
- [297] Sebastian Thrun and Lorien Pratt. *Learning to learn*. Springer Science & Business Media, 2012.
- [298] Richard Socher Tianmin Shu Caiming Xiong. “Hierarchical and Interpretable Skill Acquisition in Multi-task Reinforcement Learning”. In: *International Conference on Learning Representations*. 2018.
- [299] Emanuel Todorov, Tom Erez, and Yuval Tassa. “Mujoco: A physics engine for model-based control”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012.
- [300] Faraz Torabi, Garrett Warnell, and Peter Stone. “Behavioral cloning from observation”. In: *International Joint Conference on Artificial Intelligence*. 2018.
- [301] Faraz Torabi, Garrett Warnell, and Peter Stone. “Generative adversarial imitation from observation”. In: *arXiv preprint arXiv:1807.06158* (2018).
- [302] Eleni Triantafillou, Tyler Zhu, Vincent Dumoulin, Pascal Lamblin, Kelvin Xu, Ross Goroshin, Carles Gelada, Kevin Swersky, Pierre-Antoine Manzagol, and Hugo Larochelle. “Meta-dataset: A dataset of datasets for learning to learn from few examples”. In: *International Conference on Learning Representations*. 2020.
- [303] Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J. Lim. “Learning to Synthesize Programs as Interpretable and Generalizable Policies”. In: *Neural Information Processing Systems*. 2021.
- [304] Hado Van Hasselt, Arthur Guez, and David Silver. “Deep reinforcement learning with double q-learning”. In: *AAAI Conference on Artificial Intelligence*. 2016.
- [305] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. “Attention Is All You Need”. In: *Neural Information Processing Systems*. 2017.
- [306] Abhinav Verma, Hoang Le, Yisong Yue, and Swarat Chaudhuri. “Imitation-projected programmatic reinforcement learning”. In: *Neural Information Processing Systems*. 2019.

- [307] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. “Programmatically interpretable reinforcement learning”. In: *International Conference on Machine Learning*. 2018.
- [308] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. “FeUdal Networks for Hierarchical Reinforcement Learning”. In: *International Conference on Machine Learning*. 2017.
- [309] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. “Grandmaster level in StarCraft II using multi-agent reinforcement learning”. In: *Nature* (2019).
- [310] Oriol Vinyals, Charles Blundell, Tim Lillicrap, Daan Wierstra, et al. “Matching networks for one shot learning”. In: *Neural Information Processing Systems*. 2016.
- [311] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. “Show and tell: A neural image caption generator”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2015.
- [312] Adam Vogel and Daniel Jurafsky. “Learning to Follow Navigational Directions”. In: *Assosiation of Computational Linguistics*. 2010.
- [313] Risto Vuorio, Dong-Yeon Cho, Daejoong Kim, and Jiwon Kim. “Meta continual learning”. In: *arXiv preprint arXiv:1806.06928* (2018).
- [314] Risto Vuorio, Shao-Hua Sun, Hexiang Hu, and Joseph J. Lim. “Multimodal Model-Agnostic Meta-Learning via Task-Aware Modulation”. In: *Neural Information Processing Systems*. 2019.
- [315] Risto Vuorio, Shao-Hua Sun, Hexiang Hu, and Joseph J. Lim. “Toward Multimodal Model-Agnostic Meta-Learning”. In: *Meta-Learning Workshop at Neural Information Processing Systems*. 2018.
- [316] Catherine Wah, Steve Branson, Peter Welinder, Pietro Perona, and Serge Belongie. “The caltech-ucsd birds-200-2011 dataset”. In: (2011).
- [317] Jane X Wang, Zeb Kurth-Nelson, Dharshan Kumaran, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Demis Hassabis, and Matthew Botvinick. “Prefrontal cortex as a meta-reinforcement learning system”. In: *Nature Neuroscience* (2018).
- [318] Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. “Learning to reinforcement learn”. In: *arXiv preprint arXiv:1611.05763* (2016).
- [319] Ke Wang, Rishabh Singh, and Zhendong Su. “Dynamic neural program embedding for program repair”. In: *arXiv preprint arXiv:1711.07163* (2017).
- [320] Sida I Wang, Samuel Ginn, Percy Liang, and Christoper D Manning. “Naturalizing a programming language via interactive learning”. In: *Assosiation of Computational Linguistics*. 2017.
- [321] Tingwu Wang, Renjie Liao, Jimmy Ba, and Sanja Fidler. “Nervenet: Learning structured policy with graph neural networks”. In: *International Conference on Learning Representations*. 2018.

- [322] Ziyu Wang, Josh S Merel, Scott E Reed, Nando de Freitas, Gregory Wayne, and Nicolas Heess. “Robust imitation of diverse behaviors”. In: *Neural Information Processing Systems*. 2017.
- [323] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. “Sorting and transforming program repair ingredients via deep learning code similarities”. In: *IEEE International Conference on Software Analysis, Evolution and Reengineering*. 2019.
- [324] Ronald J Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine learning* (1992).
- [325] Elly Winner and Manuela Veloso. “DISTILL: Learning Domain-Specific Planners by Example”. In: *International Conference on Machine Learning*. 2003.
- [326] Catherine Wong, Kevin Ellis, Joshua B Tenenbaum, and Jacob Andreas. “Leveraging Language to Learn Program Abstractions and Search Heuristics”. In: *International Conference on Machine Learning*. 2021.
- [327] Jiajun Wu, Joshua B Tenenbaum, and Pushmeet Kohli. “Neural Scene De-rendering”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2017.
- [328] Da Xiao, Jo-Yu Liao, and Xingyuan Yuan. “Improving the Universality and Learnability of Neural Programmer-Interpreters with Combinator Abstraction”. In: *International Conference on Learning Representations*. 2018.
- [329] Saining Xie, Sainan Liu, Zeyu Chen, and Zhuowen Tu. “Attentional shapecontextnet for point cloud recognition”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018.
- [330] Qi Xin and Steven P Reiss. “Leveraging syntax-related code for automated program repair”. In: *International Conference on Automated Software Engineering*. 2017.
- [331] Danfei Xu, Suraj Nair, Yuke Zhu, Julian Gao, Animesh Garg, Li Fei-Fei, and Silvio Savarese. “Neural task programming: Learning to generalize across hierarchical tasks”. In: *IEEE International Conference on Robotics and Automation*. 2018.
- [332] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. “Show, attend and tell: Neural image caption generation with visual attention”. In: *International Conference on Machine Learning*. 2015.
- [333] Jun Yamada, Youngwoon Lee, Gautam Salhotra, Karl Pertsch, Max Pflueger, Gaurav S Sukhatme, Joseph J. Lim, and Peter Englert. “Motion planner augmented reinforcement learning for robot manipulation in obstructed environments”. In: *Conference on Robot Learning*. 2020.
- [334] Yujun Yan, Kevin Swersky, Danai Koutra, Parthasarathy Ranganathan, and Milad Hashemi. “Neural execution engines: Learning to execute subroutines”. In: *Neural Information Processing Systems*. 2020.
- [335] Chao Yang, Xiaojian Ma, Wenbing Huang, Fuchun Sun, Huaping Liu, Junzhou Huang, and Chuang Gan. “Imitation learning from observations by minimizing inverse dynamics disagreement”. In: *Neural Information Processing Systems*. 2019.

- [336] Yichen Yang, Jeevana Priya Inala, Osbert Bastani, Yewen Pu, Armando Solar-Lezama, and Martin Rinard. “Program Synthesis Guided Reinforcement Learning”. In: *arXiv preprint arXiv:2102.11137* (2021).
- [337] Yuxiang Yang, Ken Caluwaerts, Atil Iscen, Jie Tan, and Chelsea Finn. “NoRML: No-reward meta learning”. In: *International Conference on Autonomous Agents and Multiagent Systems*. 2019.
- [338] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. “Xlnet: Generalized autoregressive pretraining for language understanding”. In: *Neural Information Processing Systems*. 2019.
- [339] Zichao Yang, Xiaodong He, Jianfeng Gao, Li Deng, and Alex Smola. “Stacked attention networks for image question answering”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2016.
- [340] Michihiro Yasunaga and Percy Liang. “Graph-based, Self-Supervised Program Repair from Diagnostic Feedback”. In: *International Conference on Machine Learning*. 2020.
- [341] Han-Jia Ye, Hexiang Hu, De-Chuan Zhan, and Fei Sha. “Learning embedding adaptation for few-shot learning”. In: *arXiv preprint arXiv:1812.03664* (2018).
- [342] Pengcheng Yin and Graham Neubig. “Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation”. In: *Empirical Methods in Natural Language Processing*. 2018.
- [343] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. “Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task”. In: *Empirical Methods in Natural Language Processing*. 2018.
- [344] Tianhe Yu, Chelsea Finn, Annie Xie, Sudeep Dasari, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. “One-Shot Imitation from Observing Humans via Domain-Adaptive Meta-Learning”. In: *Robotics: Science and Systems*. 2018.
- [345] Tianhe Yu, Aviral Kumar, Rafael Rafailov, Aravind Rajeswaran, Sergey Levine, and Chelsea Finn. “Combo: Conservative offline model-based policy optimization”. In: *arXiv preprint arXiv:2102.08363* (2021).
- [346] Wojciech Zaremba and Ilya Sutskever. “Reinforcement learning neural turing machines-revised”. In: *arXiv preprint arXiv:1505.00521* (2015).
- [347] Grace Zhang, Linghan Zhong, Youngwoon Lee, and Joseph J. Lim. “Policy Transfer across Visual and Dynamics Domain Gaps via Iterative Grounding”. In: *Robotics: Science and Systems*. 2021.
- [348] Han Zhang, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. “Self-Attention Generative Adversarial Networks”. In: *International Conference on Machine Learning*. 2019.
- [349] Jesse Zhang, Brian Cheung, Chelsea Finn, Sergey Levine, and Dinesh Jayaraman. “Cautious Adaptation For Reinforcement Learning in Safety-Critical Settings”. In: *International Conference on Machine Learning*. 2020.

- [350] Zelin Zhao, Karan Samel, Binghong Chen, and Le Song. “ProTo: Program-Guided Transformer for Program-Guided Tasks”. In: *Neural Information Processing Systems*. 2021.
- [351] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. “An inductive synthesis framework for verifiable reinforcement learning”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019.
- [352] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, and Anind K Dey. “Maximum entropy inverse reinforcement learning”. In: *AAAI Conference on Artificial Intelligence*. 2008.
- [353] Luisa Zintgraf, Kyriacos Shiarli, Vitaly Kurin, Katja Hofmann, and Shimon Whiteson. “Fast context adaptation via meta-learning”. In: *International Conference on Machine Learning*. 2019.
- [354] Luisa Zintgraf, Kyriacos Shiarlis, Maximilian Igl, Sebastian Schulze, Yarin Gal, Katja Hofmann, and Shimon Whiteson. “VariBAD: A Very Good Method for Bayes-Adaptive Deep RL via Meta-Learning”. In: *International Conference on Learning Representations*. 2020.
- [355] Konrad Zolna, Scott Reed, Alexander Novikov, Sergio Gomez Colmenarej, David Budden, Serkan Cabi, Misha Denil, Nando de Freitas, and Ziyu Wang. “Task-Relevant Adversarial Imitation Learning”. In: *Conference on Robot Learning*. 2020.