Neurosymbolic Learning for Robust and Reliable Intelligent Systems

by

Jeevana Priya Inala

B.S., Massachusetts Institute of Technology (2016)
M.Eng., Massachusetts Institute of Technology (2016)

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

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2022

© Massachusetts Institute of Technology 2022. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
September 29, 2021

Certified by ....
Armando Solar-Lezama
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

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

Neurosymbolic Learning for Robust and Reliable Intelligent Systems

by

Jeevana Priya Inala

Submitted to the Department of Electrical Engineering and Computer Science
on September 29, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

This thesis shows that looking at intelligent systems through the lens of neurosymbolic models has several benefits over traditional deep learning approaches. Neurosymbolic models contain symbolic programmatic constructs such as loops and conditionals and continuous neural components. The symbolic part makes the model interpretable, generalizable, and robust, while the neural part handles the complexity of the intelligent systems. Concretely, this thesis presents two classes of neurosymbolic models—state-machines and neurosymbolic transformers and evaluates them on two case studies—reinforcement-learning based autonomous systems and multi-robot systems. These case studies showed that the learned neurosymbolic models are human-readable, can be extrapolated to unseen scenarios, and can handle robust objectives in the specification. To efficiently learn these neurosymbolic models, we introduce neurosymbolic learning algorithms that leverage the latest techniques from machine learning and program synthesis.

Thesis Supervisor: Armando Solar-Lezama
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like first to thank my advisor Armando Solar-Lezama. I had the privilege of working with Armando since my freshman year in undergrad and continued to work with him through grad school. I should say that everything I learned about research is from him. He always encouraged and supported me despite initial setbacks in my research. His super positive attitude always helped keep up the spirits in our lab. With Armando, I can go into a meeting with a very vague abstract idea and come out with a well-executable concrete idea. In addition to research, Armando gives the best advice on presenting and explaining an idea well; he helped me a great deal with networking, finding collaborators and ultimately helped me with my job talk. I am fortunate to have him as my advisor and mentor for the past nine years.

I want to thank my thesis committee, Mike Carbin and Josh Tenenbaum, for their insightful questions and comments on my thesis and for encouraging me to try neurosymbolic transformers in many other domains.

Next, I want to thank all of my collaborators. I am grateful to have worked with many smart people from many different institutions. A special shout out to Osbert Bastani for being my unofficial mentor. We have collaborated on multiple projects in the last few years, including the works that this thesis is based on. I learned a great deal from him about scoping out manageable projects, figuring out the suitable experiments to run, and, more importantly, writing papers. Evan Pu and Yichen Yang are some of my other frequent collaborators. I greatly enjoyed discussing and brainstorming ideas with this group of collaborators. I am fortunate to collaborate with Hadas Kres-Gazit and Thais Campos from Cornell University. Hadas's detailed feedback on both the research and the technical writing has greatly helped me grow as a researcher. We had another fun collaboration with the graphics group at MIT with Wojciech Matusik, Tao Du, and others. I also had the opportunity to intern at multiple industries; Rishabh Singh, my mentor when I interned at Microsoft Research, has been very supportive of my research and my go-to person to ask for advice on industry research.

I am grateful for being part of the CAP research group and the larger MIT CSAIL community. I collaborated with several people from the CAP group and had an opportunity to interact with almost all of them. We had lots of fun over the years (especially the coffee trips)!

Last, I owe this thesis to my family, without whose constant support this thesis would not be the same. First and foremost, I want to thank my mother for her sacrifices and continuous support. She is always the one helping my brother and me find different opportunities to succeed in life. In fact, without her networking skills, I wouldn't have applied to MIT for my undergraduate studies. Next, I want to thank my father for always being there and travelling with me to all my science camps. I am grateful for my brother, who explored the world three years ahead of me and helped make my life better with his experiences. I want to thank my sister-in-law for being my best friend, supporter, and adviser. Finally, I am grateful for having two little nieces who can relieve my stress in seconds with their cuteness!

Contents

1 Introduction **19**

1.1 Neurosymbolic Models 21

1.2 Neurosymbolic Learning 21

1.3 Contributions 22

1.3.1 Classes of Neurosymbolic Models 22

1.3.2 Neurosymbolic Learning Algorithms 24

1.3.3 Case-studies 26

2 Related Works **33**

2.1 Program Synthesis 33

2.2 Intersection of Program Synthesis and Machine Learning 34

2.3 Hybrid Systems 37

2.4 Teacher-Student Algorithms in Other Settings 37

2.5 Interpreting/Analyzing Neural Networks 38

2.6 Meta-Learning 38

3 Neurosymbolic Models **39**

3.1 State Machines 39

3.1.1 Formalism 40

3.1.2 Discussion 42

3.2 Neurosymbolic Transformers 43

3.2.1 Transformers 43

3.2.2 Attention Programs 44

3.2.3 Combining Transformers with Attention Programs 46
3.2.4 Multiple Attention Layers 47
3.2.5 Discussion 48

4 Neurosymbolic Reinforcement Learning 49
4.1 Background on Reinforcement Learning 49
4.2 Problem Formulation 50
4.3 Neurosymbolic Policies 52
4.3.1 State Machines for Tracking Internal State 52
4.3.2 Neurosymbolic Transformers for List of States 52

5 Neurosymbolic Learning Algorithms 53
5.1 Imitation Learning 54
5.2 Imitation Learning for Neurosymbolic Transformers 55
5.2.1 Optimizing Combinatorial Objectives 57
5.2.2 Program Synthesis for Supervised Learning 57
5.2.3 Discussion 58
5.3 Shortcomings of Imitation Learning 59
5.4 Adaptive Teaching 59
5.4.1 Overview 60
5.4.2 Adaptive Teaching via Variational Inference 60
5.5 Instantiating Adaptive Teaching for Learning State Machine Policies . 63
5.5.1 Trace Parameterization 63
5.5.2 Teacher's Optimization 64
5.5.3 Student's Imitation Learning 64
5.5.4 Example 66
5.5.5 EM Approach Details 67
5.5.6 Synthesizing Switching Conditions 70
5.5.7 Discussion 72

6 Case Study — State Machine Policies for Reinforcement Learning Control Tasks **73**
6.1 Tasks . . . 74
6.2 Baselines . . . 77
6.2.1 RL Baselines . . . 77
6.2.2 Direct-Opt Baseline . . . 79
6.3 Hyper-Parameters in Adaptive Teaching . . . 80
6.4 Emperical Results . . . 81
6.4.1 Inductive Generalization . . . 81
6.4.2 Interpretability . . . 85
6.4.3 Verification . . . 86
6.4.4 Behavior of Policy . . . 87
6.4.5 Analysis of Running Time . . . 88
6.5 Discussion . . . 89

7 Case Study: Neurosymbolic Transformers for Multi-Agent Communications **93**
7.1 Multi-Agent RL Related work . . . 95
7.2 Multi-Agent Problem Formulation . . . 96
7.3 Neurosymbolic Transformers for Multi-Agent problem . . . 98
7.4 Tasks . . . 99
7.5 Baselines . . . 100
7.6 Hyper-parameters . . . 102
7.7 Emperical Results . . . 103
7.7.1 Performance and Expressiveness . . . 103
7.7.2 Combinatorial Optimization—Reducing the Communication Degree . . . 107
7.7.3 Interpretability . . . 108
7.7.4 Generalization and Robustness . . . 108
7.8 Discussion . . . 111

8 Future Directions **113**

List of Figures

1-1 An example of an autonomous car (blue) driving out of a parallel parked spot. The trajectories are obtained using our learned neurosymbolic model. 26

1-2 A learned DNN policy fails to generalize for the car task. It solves the training tasks but runs into collisions on the test task. 27

1-3 A learned state machine policy for the task in Figure 1-1. The boxes are the three different modes. Each mode performs a simple action. Switching conditions (arrows) decide when the state machine switches from one mode to another. Trivially false switching conditions are dropped. The state machine starts in mode m_s and ends in mode m_e. 28

1-4 (a) Three groups of agents (blue, green, and red) at their initial positions (circles) trying to reach their goal positions (crosses). The solid line shows the trajectory taken by a single agent in each group. (b) Soft attention computed by a DNN for the agent along the y-axis deciding whether to focus on the agent along the x-axis. (c) Sparse attention computed by a program. (d) Program used by each agent to select other agents to focus on (linear functions are abstracted for simplicity). $\langle x, y \rangle$ denotes dot product, θ denotes learned parameters, and ϕ denotes the feature vector. 29

1-5 Visualization of the programmatic attention layer in Figure 1-4d, which has two rules R_1 and R_2. In this task, there are three groups of agents. The red circle denotes the agent currently choosing an action, the red cross denotes its goal, and the green circle denotes the agent selected by the rule. (a,b) Visualization of rule R_1 for two different states; orange denotes the region where the filter condition is satisfied—i.e., R_1 chooses a random agent in this region. (c) Visualization of rule R_2, showing the scores output by the map operator; darker values are higher—i.e., the rule chooses the agent with the darkest value. . . . 32

3-1 Neural networks vs state machine models — a conceptual comparison. 41

3-2 A traditional transformer's soft attention architecture vs an attention program. 46

5-1 High-level overview of the adaptive teaching algorithm. 60

5-2 An example loop-free policy. This policy has a sequence of 4 simple policies. The grammar for simple policies, in this case, is scalar constants (same as the state machine policy in Figure 1-3). Each simple policy H_i is executed for T_i duration before switching to the next policy. 63

5-3 The high-level overview of the student's EM approach to imitate the teacher for learning state machine policies. 66

5-4 Visualization showing the student-teacher interaction for two iterations. (a) The loop-free policies (with their corresponding rewards) learned by the teacher for two different initial states. Here, the boxes signify the different segments in the loop-free policies, the colors signify different actions, and the boxes' lengths signify the segments' durations. (b) The mapping between the segments and the modes in the state machine—i.e., $p(\mu = m_j)$. Each box shows the composition of the modes vertically distributed according to their probabilities. For example, the third segment in the loop-free policy for x_0^1 has $p(\mu = \text{Green}) = 0.65$ and $p(\mu = \text{Brown}) = 0.35$. (c) The most probable rollouts from the state machine policy learned by the student. Finally, (d), (e) and (f) are similar to (a), (b) and (c), but for the second iteration. 67

5-5 Switching conditions represented as decision trees. 71

6-1 Summary of our benchmarks. #A is the action dimension, #S is the state dimension, X_0^{train} is the set of initial states used for training, X_0^{test} is the set of initial states used to test inductive generalization, # modes is the number of modes in the state machine policy, and M_G and C_G are the grammars for the simple functions in the modes and the switching conditions, respectively. Depth of C_G indicates the number of levels in the Boolean tree. 75

6-2 Trajectories for the Quad (left) and QuadPO (right) benchmarks using our state machine policy. 75

6-3 Comparison of performances on the train distribution. Our approach performs almost similar to the RL baselines, showing that our approach is expressive for these tasks. An empty bar indicates that the policy learned for that experiment failed on all runs. 81

6-4 Comparison of performances on the test distribution. Our approach outperforms the baselines on all benchmarks in terms of test performance. An empty bar indicates that the policy learned for that experiment failed on all runs. 82

6-5 Experiment results for additional benchmarks. G is the average goal error (closer to 0 is better). T_G is the average number of timesteps to reach the goal (lower the better). ⊥ indicates a timeout. We can see that both our approach and RL generalizes for these benchmarks. 82

6-6 Trajectories taken by our state machine policy (left) and the RL policy (right) on Pendulum for a test environment (i.e., heavier pendulum). Green (resp., red) indicates positive (resp., negative) torque. Our policy performs optimally by using positive torque when angular velocity ≥ 0 and negative torque otherwise. In contrast, the RL policy performs sub-optimally (especially at the beginning of the trajectory). 83

6-7 Trajectories taken by our state machine policy on Swimmer for (a) a train environment with segments of length 1, and (b) a test environment with segments of length 0.75. The colors indicate different modes. The axes are the x and y coordinates of the center of mass of the Swimmer. Trajectories taken by the RL policy on Swimmer for (c) a train environment, and (d) a test environment. While both policies generalize, the Swimmer with the state machine policy is slightly faster (it takes about 35s to cover a distance of 10 units while the RL policy takes about 45s). 83

6-8 The RL policy generates unstructured trajectories, and therefore does not generalize from (a) the training distribution to (b) the test distribution. In contrast, our state machine policy in (c) generates a highly structured trajectory that generalizes well. 84

6-9 We plot the train and the test performance for different choices of training distribution for the Car benchmark 85

6-10 A user can modify a learned state machine policy to improve performance. In (b), the user sets the steering angle in Figure 1-3 to the maximum value 0.5, and in (c), the user sets the thresholds in the switching conditions $G_{m_1}^{m_2}, G_{m_2}^{m_1}$ to 0.1. 86
6-11 A failure case found with verification with a noise of 0.24 in the environment, where the car collides with the car in the front. 87
6-12 Graph of vertical acceleration over time for both our policy (red) and the neural network policy (blue) for Quad (left) and QuadPO (right). 88
6-13 Action vs time graphs for the car benchmark for our policy (red) and the neural network policy (blue). (Left) shows the velocity of the agent, and (Right) shows the steering angle. 88
6-14 Action vs time graphs for the pendulum benchmark (left) and the cartpole benchmark (right) for both our policy (red) and the neural network policy (blue). 89
6-15 Action vs time graphs for the swimmer benchmark for the three torques at the three different joints of the swimmer. The blue line is for the neural network policy, and the red is for the state machine policy. . . 89
6-16 Synthesis times (in seconds, wall clock time) for learning state machines policies for the different benchmarks. The plot breaks down the total synthesis time into the time taken by the teacher, the student and other miscellaneous parts of the algorithm. Misc. mainly includes the time spent for checking convergence at every iteration. The plot also shows the number of teacher-student iterations taken for each benchmark. . 90
6-17 Synthesized state machine policy for the Quad benchmark. 90
6-18 Synthesized state machine policy for the QuadPO benchmark. 90
6-19 Synthesized state machine policy for Pendulum. 90
6-20 Synthesized state machine policy for Cartpole. 91
6-21 Synthesized state machine policy for Acrobot. 91
6-22 Synthesized state machine policy for Mountain car. 91
6-23 Synthesized state machine policy for Swimmer. 91

7-1 Unlabeled goals task: (a) Initial positions of the agents and the locations of the goals to cover (b) Final configuration of the agents where 8 out of the 10 goals are covered. 100

7-2 Statistics of cumulative loss and communication graph degrees across baselines, for (a) random-cross, (b) random-grid, and (c) unlabeled-goals. We omit communication degrees for `tf-full`, since it requires communication between all pairs of agents. 104

7-3 For `random-cross`, trajectories taken by each group (i.e., averaged over all agents in that group) when all four groups are present (left) and only one group is present (right), by `prog-retrained` (solid) and `dist` (dashed). Initial positions are circles and goal positions are crosses. 105

7-4 Comparing attention program with a RL policy that treats communications as actions. `RL1` and `RL2` correspond to two different hyperparameters in the policy gradient algorithm that achieved lowest loss and lowest communication degree (respectively). 106

7-5 Attention weights for `hard-attn` and `prog-retrained` at a single step near the start of a rollout, computed by the agent along the y-axis for the message from the agent along the x-axis. 107

7-6 Attention maps of `prog-retrained` for the two rounds of communication for the unlabeled goals task. 108

7-7 For the `random-grid` task, we show how the transformer policy adapts to new settings. On the left, we show the trajectories for a setting from the train distribution, i.e. with an equal number of agents in all groups. On the right, we show the trajectories for a new setting from a different test distribution, i.e. with an asymmetric number of agents in the groups. We see that a transformer policy has an undesirable side-effect; it pays more attention to the group with more agents. 109

7-8 For the `random-grid` task, we show how the neurosymbolic transformer policy adapts to new settings. On the left, we show the trajectories for a setting from the train distribution, i.e. with an equal number of agents in all groups. On the right, we show the trajectories for a new setting from a different test distribution, i.e. with an asymmetric number of agents in the groups. We see that the prog-retrained policy correctly attends to all groups (irrespective of the number of agents in a group). 110
7-9 Random grid task with noisy communications. 110

Chapter 1

Introduction

Machine learning-based artificial intelligence (AI) systems are prevalent in almost all fields, including autonomous cars, robotics, medicine, and finance. Most of these domains are safety-critical, and hence, it is costly for a machine learning-based system to make mistakes. Therefore, the AI systems of the future need to be robust, reliable, and trustworthy. Below are some essential characteristics that make a AI system robust and reliable:

- **Interpretability**: An interpretable AI system allows a user/a developer to understand why the system made a particular decision/prediction, probe the system for corner cases, and even modify the system to make small changes without training the system from scratch.

- **Generalization**: Being able to generalize to new scenarios is a core requirement for any intelligent system. But even within generalization, there are two types—generalization that requires (i) interpolation and (ii) extrapolation. The first kind is relatively easy, and many current machine-learning systems already achieve that, but the latter type is challenging and crucial. Extrapolation allows us to learn an intelligent system from fewer data, train a system in simulation and yet, use it in the real world, and so on.

- **Ability to handle combinatorial objectives**: A developer of an AI system specifies their desired intent through an objective, which is then optimized to

learn the best AI system. E.g. one typical objective is to minimize the L2 loss between the predicted output and the actual output summed over all data points. In most cases, these objectives are smooth, continuous, and convex, making the optimization problem easier. But, there are also scenarios where the desired objective is combinatorial. Examples of combinatorial objectives occur commonly in problems that involve routing or multi-agent systems [12]. Hence, being able to handle combinatorial objectives is an important criterion.

- **Verifiability**: Finally, formally verifying that an intelligent system satisfies the desirable constraints such as safety, robustness, and fairness strongly reinforces reliability and trustworthiness.

The state-of-the-art for many of the current AI systems uses deep learning approaches to train a complex deep neural network (DNN) for the task at hand. However, traditional deep learning approaches lack the above desirable properties needed for a robust and reliable system. That said, recently, the deep learning community has been working hard to achieve the above properties; In this space, there are works that can visualize the learned features of a convolutional neural network [70] to aid interpretability, that can do reachability analysis of neural networks and use that to verify some properties of the network [51, 37], and that can analyze the robustness of the networks, discover adversarial examples, and perform adversarial training [21, 38, 63, 84, 7]. However, reasoning about complex unstructured neural networks is fundamentally a challenging problem.

This thesis explores a different approach that allows us to achieve the above four properties more directly. The high-level idea is to reduce the complexity of the models using symbolic/programmatic structures—we call these types of models as *neurosymbolic models*. Then, we attempt to answer the following questions:

- What kinds of neurosymbolic models can we learn that has the above desirable properties for robustness and reliability?

- How can we efficiently learn these neurosymbolic models?

1.1 Neurosymbolic Models

Many real-world objects such as buildings and roads, and human activities such as walking, swimming, and driving are very structured— e.g. they have many repeating units. This observation motivates us to learn models for the intelligent systems that can explicitly capture structures such as repetitions, compositions, and logical relations. And, by doing so, we learn models that are easy to interpret, generalize better and hence, are more robust and reliable.

How can we capture these structures? The answer is symbolic models or programs. Software developers have long been using programs to communicate with a computer in a structured format. The *loop* concept in programs is used to represent repetitions, *functions* are used for modularity and compositionality, and *Boolean logic* and *branches* are used to capture logical relations between different entities. These discrete programming concepts are not restricted to just software engineering tasks and are one of the best candidates to model the structure in intelligent systems.

However, in addition to the structure, real-world intelligent systems have noise, uncertainty, and continuous components. To handle these, we will need continuous models like affine/linear models or neural networks.

In this thesis, we use the term neurosymbolic models to represent this combination of discrete programming concepts (such as loops, branches, and functions) and continuous concepts (such as neural networks and linear models).

Note that the idea here is not to throw away deep learning and go back to a purely symbolic AI era but to leverage all the new things we can do with both programs and neural networks to get the best of both worlds.

1.2 Neurosymbolic Learning

While neurosymbolic models have several benefits, learning them is very challenging due to the highly discrete-continuous search space. As a result, we cannot just use machine learning/deep learning techniques, which mainly use gradient-based numer-

ical optimization. These techniques can efficiently handle the continuous structure but not the discrete part. On the other hand, there is program synthesis—a field dedicated to synthesizing programs from specification [91, 41, 3]. However, traditional program synthesis mainly handles only discrete programs—these techniques use the many well-known discrete search algorithms like SAT solving or enumeration. Program synthesis can handle the discrete structure but cannot handle the continuous structure.

Thus, the idea of neurosymbolic learning is to leverage the techniques from both machine learning and program synthesis to get the best of both worlds and efficiently learn the neurosymbolic models.

1.3 Contributions

This thesis makes three kinds of contributions: (1) Developing two classes of neurosymbolic models as replacements to purely deep neural models (Chapter 3). (2) Developing neurosymbolic learning algorithms that can efficiently learn models in the above classes of models (Chapter 5), and (3) Finally evaluating the efficiency of these neurosymbolic learning/models on two different case-studies (Chapters 6 and 7) in reinforcement learning (Chapter 4).

1.3.1 Classes of Neurosymbolic Models

The machine learning community has already explored a few interpretable models such as decision trees [15] and rule lists [103]. In particular, these models can be thought of as simple programs composed of simple primitives such as if-then-else rules and arithmetic operations. However, a key shortcoming of these model classes is that they have difficulty handling more complex inputs, e.g., sets of other inputs or sequences of inputs. Thus, one of the contributions of this thesis is to introduce two classes of neurosymbolic models with more sophisticated components.

State Machines

One fundamental limitation of decision trees and rule lists is that they do not possess an internal memory. Internal memory helps propagate information about the current iteration to the next when processing a sequence of inputs (such as processing a text) or during sequential decision-making (such as reinforcement learning tasks). In the context of deep learning, one can use recurrent neural networks (RNNs) and long-short term memory networks (LSTMs) to store internal memory.

In the symbolic domain, one natural analog is to use models based on finite state machines. So, we designed a class of neurosymbolic state machine models [46]. Our *state machine models* are designed to be interpretable, generalizable, and verifiable while including internal memory. Its internal state records one of a finite set of possible *modes*, each of which is annotated with (i) a simple model for computing the current output when in this mode (e.g., a linear function of the input), and (ii) rules for when to transition to the next mode (e.g., if some linear inequality becomes satisfied, then transition to a given next mode). With this single integer of internal memory, state machines can encode complex non-linear logic compactly (e.g. repetitions can be encoded as a sequence of modes connected as a loop). Although state machines by themselves are not a new idea, in this thesis, we view them as a class of models for representing intelligent systems and present a efficient learning algorithm for this class.

Neurosymbolic Transformers

Handling a list of elements or a variable number of elements as input is another difficulty for traditional symbolic models such as decision trees. For example, in multi-robot systems, the full input consists of a list of states for each individual robot. In this case, the model must compute a single action for the robot based on the given list of states. Alternatively, for problems with variable numbers of objects, the set of object positions must be encoded as a list.

In the deep learning domain, transformer-based architectures have emerged to

solve these kinds of tasks. At a high level, a transformer [98] is a DNN that operates on a list of elements. A transformer first chooses a small subset of other elements of the list to focus on (the attention layer), then uses a fully-connected layer to decide what information from the other elements is useful (the value layer), and finally uses a second fully connected layer to compute the result (output layer). For example, transformers can be applied to multi-robot systems since they have to reason over lists of other robots' states.

In the symbolic world, an analog we explored in this thesis is list processing programs, which are compositions of components designed to manipulate lists—e.g., the map, filter, and fold operators [33]; the set of possible components can be chosen based on the application. In fact, we leverage these list processing programs only to replace the attention mechanism for a transformer model. This combination gives rise to a *neurosymbolic transformer* [47], which is similar to a transformer but uses attention programs instead of deep neural attention layers; the value layer and the output layer are still neural networks. This architecture makes the attention layer interpretable—e.g., it is easy to understand and visualize why an element attends to another element while still retaining much of the complexity of the original transformer.

1.3.2 Neurosymbolic Learning Algorithms

Our next contribution is algorithms for training these neurosymbolic models.

Imitation Learning

Imitation Learning is a standard approach where we first train a deep neural network (DNN) model using deep learning. Then, using this DNN as a teacher/oracle, we train a student (in this case, the neurosymbolic model) to imitate the DNN. This approach is especially useful in scenarios where the original learning problem is not a simple supervised learning problem, such as e.g. reinforcement learning (RL) problems. In RL settings, the model (also called a policy) needs to output sequences of highly connected decisions, and the loss function (the reward) is typically sparse (e.g. only

at the end of the trajectory). In such cases, this imitation learning approach can leverage sophisticated deep learning approaches developed for these settings (e.g. policy gradient algorithm for RL problems) to train the teacher efficiently. Then, the algorithm uses this teacher to turn the student's learning problem into a direct supervised learning problem, which is much easier to optimize.

In our experiments, we found this simple strategy of imitation learning to be a good fit for learning neurosymbolic transformers. Here, the teacher learns a deep neural transformer model to solve the problem (e.g., maximising the RL problems' reward). Then, using the teacher model, we gather a supervised dataset of input-attention-output pairs. Next, we use program synthesis algorithms (such as a stochastic MCMC search algorithm) to search in the symbolic space of the attention programs to find one that achieves a lower loss on the supervised dataset. Now, since we are already doing a combinatorial search over the space of programs, it is easier to include any combinatorial objectives that the task requires.

Adaptive Teaching

One key drawback of imitation learning is that it does not adjust the teacher model to account for the capabilities of the student model. For e.g. a teacher for a state machine model needs to account that the state machine can only perform a limited number of different logics (limited by the number of modes in the state machine model).

So, we propose a new approach called *adaptive teaching* [46], where rather than choosing the teacher to be a DNN, it is instead a model whose structure mirrors that of the student. In this case, we can directly update the teacher on each training iteration to reflect the structure of the student. For example, for the state machine models, we chose the teacher to be a "loop-free" model, which consists of a linear sequence of modes (instead of modes connected by switching conditions). These modes can then be mapped to the modes of the student state machine and regularized so that their local logics and mode transitions mirror that of the state machine. This approach regularizes the teacher to favor strategies similar to the ones taken by the student to

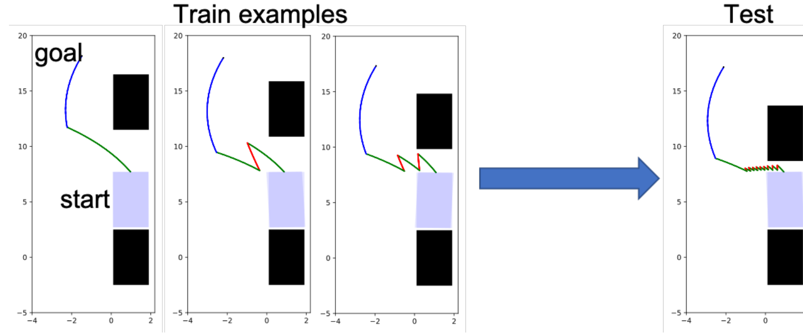


Figure 1-1: An example of an autonomous car (blue) driving out of a parallel parked spot. The trajectories are obtained using our learned neurosymbolic model.

ensure the student can successfully mimic the teacher. As the student improves, the teacher improves as well. We found adaptive teaching to be an effective strategy for learning state machine models.

1.3.3 Case-studies

The neurosymbolic learning approach is applicable in a wide range of domains; in particular, this thesis focuses on reinforcement learning applications, where reliability, robustness, and the ability to enforce constraints are essential for safety. This thesis focuses on the following two case studies:

Learning Interpretable, Generalizable, and Verifiable Policies for Control Tasks

Generalization is essential in robots. Autonomous systems must possess the capacity to work in various environments, including environments never seen previously. To achieve this generalization, autonomous systems have to learn to extrapolate from the scenarios seen before. E.g., consider the autonomous car (blue) in Figure 1-1, whose goal is to move out of a parallel parking spot. From the first three scenarios, it is clear that the car needs to make repetitive back-and-forth motions to exit the parking spot. A policy that can capture this repetition logic can extrapolate to even scenarios where the gap between the cars is tiny (something that is not encountered before).

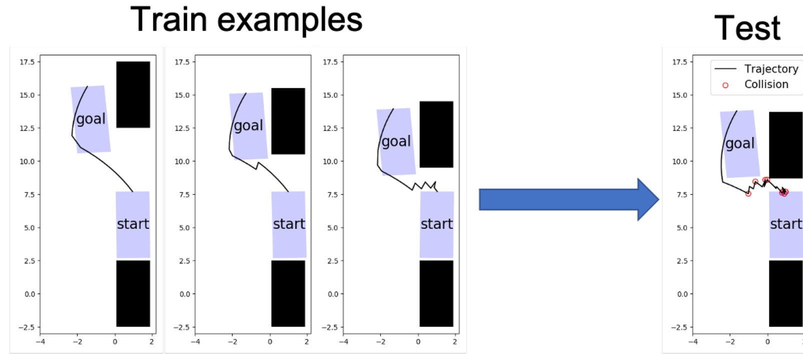


Figure 1-2: A learned DNN policy fails to generalize for the car task. It solves the training tasks but runs into collisions on the test task.

In addition to generalization, having an interpretable policy for an autonomous car allows human experts to understand and debug various behaviors of the car and even modify the policy to adapt to different situations. For example, in the above example, we want an expert to be able to adapt the policy to a different car that has different maximum acceleration and maximum steering angle capacities without having to do the expensive training from scratch.

Another critical challenge in many real-world applications is the need to ensure that the learned policy for an autonomous agent continues to act correctly once it is deployed in the real world. However, DNN policies are typically very difficult to understand and analyze, making it hard to have guarantees about their performance. The reinforcement learning setting is particularly challenging since we need to reason not just about isolated predictions but about sequences of highly connected decisions. For example, in the above scenario, we would like to verify that a learned policy would eventually get the car out without any collisions for some desired settings under some target noise model for the car's behavior.

We cast the control learning problem as a neurosymbolic learning problem to learn such generalizable, interpretable, and verifiable behaviours [46]. In our approach, the neurosymbolic control policies are modeled as state machines that can capture the repetitive structures necessary for extrapolation. On a set of reinforcement learning (RL) tasks involving cars, quadcopters, cartpoles, and pendulums, we showed that while traditional deep reinforcement learning approaches perform well on the original

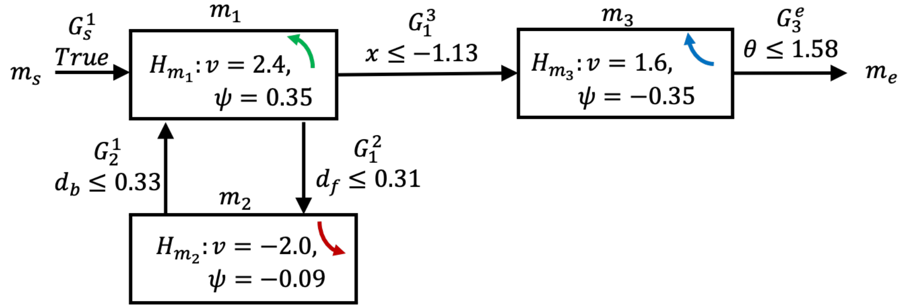
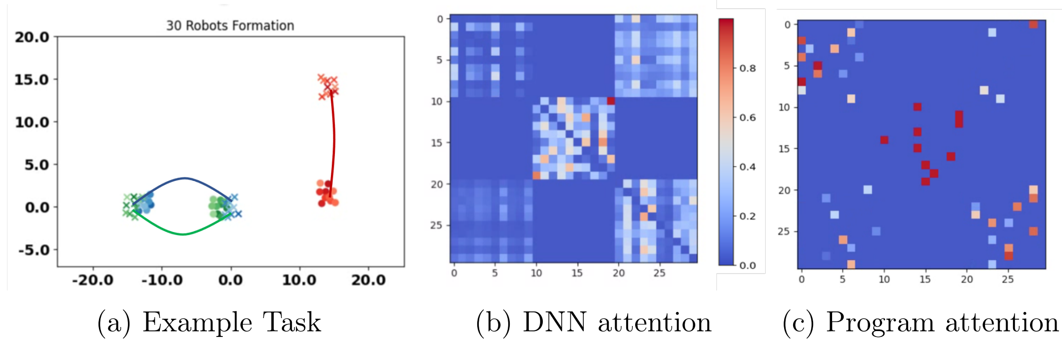


Figure 1-3: A learned state machine policy for the task in Figure 1-1. The boxes are the three different modes. Each mode performs a simple action. Switching conditions (arrows) decide when the state machine switches from one mode to another. Trivially false switching conditions are dropped. The state machine starts in mode m_s and ends in mode m_e.

task, they fail to generalize (e.g. see Figure 1-2). In contrast, our neurosymbolic policies successfully generalize beyond the training distribution (as shown in Figure 1-1).

Figure 1-3 shows the learned state machine policy for the task in Figure 1-1. The state of the car is $(x, y, \theta, d_f, d_b) \in \mathbb{R}^5$, where (x, y) is the center of the car, θ is its orientation, and d_f and d_b are the distances between the agent and the front and back black cars, respectively. The actions are $(v, \psi) \in \mathbb{R}^2$, where v is the velocity and ψ is the steering angle. This policy has three modes (besides a start mode m_s and an end mode m_e). Roughly speaking, it says (i) immediately shift from mode m_s to m_1, and drive the car forward and to the left, (ii) continue until close to the car in front; then, transition to mode m_2, and drive the car backwards and to the right, (iii) continue until close to the car behind; then, transition back to mode m_1, (iv) iterate between m_1 and m_2 until the car can safely exit the parking spot; then, transition to mode m_3, and drive forward and to the right to make the car parallel to the lane. Note that, here, the two modes m_1 and m_2 are connected in a loop, and this is how the policy can represent the repetitive back and forth motion needed for this task.

Furthermore, since the state machine policy has simple logic inside each mode (e.g. scalar constants or linear functions), they are easy to interpret and modify. For example, adapting to a car with wider steering angle ranges requires replacing only three scalars in the learned policy with new values. Moreover, the discrete structure



(a) Example Task (b) DNN attention (c) Program attention

$$R_1 : \mathsf{random}(\mathsf{filter}(\langle \theta_1, \phi \rangle \geq \theta_1^0, \ell)),$$
$$R_2 : \mathsf{argmax}(\mathsf{map}(\langle \theta_3, \phi \rangle, \mathsf{filter}(\langle \theta_2, \phi \rangle \geq \theta_2^0, \ell))).$$

(d) Programmatic attention rules

Figure 1-4: (a) Three groups of agents (blue, green, and red) at their initial positions (circles) trying to reach their goal positions (crosses). The solid line shows the trajectory taken by a single agent in each group. (b) Soft attention computed by a DNN for the agent along the y-axis deciding whether to focus on the agent along the x-axis. (c) Sparse attention computed by a program. (d) Program used by each agent to select other agents to focus on (linear functions are abstracted for simplicity). $\langle x, y \rangle$ denotes dot product, θ denotes learned parameters, and ϕ denotes the feature vector.

in a state machine makes it much more amenable to formal verification using off-the-shelf solvers such as dReach [55].

Learning Robust and Interpretable Multi-agent Communications with Combinatorial Objectives

To learn the control for robots, we have to satisfy the resource limitations of the physical hardware. This problem is significant in decentralized multi-agent planning domains, where the agents/robots have to communicate with the other agents to coordinate their actions. Here, the goal is to learn how to coordinate with the other agents, both deciding whom to communicate with and what information to share, and at the same time, minimizing the amount of communication required to satisfy the limited network bandwidth constraint. This minimum communication objective is a combinatorial objective which makes training DNN policies hard for this task.

For example, consider the task in Figure 1-4a, where there are three groups of

agents (blue, green, and red), and they are moving towards their respective goals placed randomly on the 3x3 grid. The agents need to communicate with the other groups to ensure that their paths do not collide. In the specific scenario shown in Figure 1-4a, the blue and the green groups need to coordinate and go around each other (instead of going straight to their goals) to avoid collisions. Here, Figure 1-4b shows the attention graph (i.e. which agents need to communicate to which other agents) computed by a DNN based transformer model. As we can see, this DNN model learns that every agent needs to communicate with almost 20 different agents, which is not optimal.

In this thesis, we propose to learn neurosymbolic transformers to represent multi-agent policies [47]. A neurosymbolic transformer uses a program instead of a neural network to compute which agents need to attend (communicate) to which other agents. Our domain-specific language (DSL) for programmatic attention includes components such as filter, map, and random choice. These components operate over sets of inputs because choosing whom to attend to requires reasoning over sets of other agents—e.g., to avoid collisions, an agent must attend to its nearest neighbor in its direction of travel.

We successfully used the neurosymbolic transformer approach on several multi-agent planning tasks that require agents to coordinate to achieve their goals. Our algorithm learns communication policies that achieve task performance similar to the original transformer policy (i.e., where each agent communicates with every other agent) while significantly reducing the amount of communication (the combinatorial objective).

Figure 1-4d shows a learned attention program that each agent can use to choose other agents to focus on. In particular, this program consists of two rules, each of which selects a single agent to focus on; the program returns the set consisting of both selected agents. In each of these rules, agent i is selecting over other agents j in the list ℓ. The first rule starts by filtering the agents in ℓ using a learned linear inequality ($\langle \theta_1, \phi \rangle \geq \theta_1^0$) where $\theta_1 \in \mathbb{R}^k$ and $\theta_1^0 \in \mathbb{R}$ are the learned parameters, $\phi \in \mathbb{R}^k$ is a vector of features involving the deciding agent's and the other agent's current

locations and goals, and $\langle x, y \rangle$ denotes a dot product. Then the rule picks a random agent in this filtered set to attend to. The second rule starts by filtering the agents in ℓ using another learned linear inequality ($\langle \theta_2, \phi \rangle \geq \theta_2^0$). Then the rule computes a score for each of the filtered agents using map and the learned scoring function $\langle \theta_3, \phi \rangle$ and finally, chooses the agent with the maximum score.

The specific learned map functions and filter conditions are visualized in Figure 1-5. The red circle denotes the agent currently choosing an action in these figures, the red cross denotes its goal, and the green circle denotes the agent selected by the rule. Figures 1-5a and 1-5b correspond to the rule R_1 for two different states. The orange region denotes where the filter condition is satisfied–i.e. R_1 chooses a random agent in this region. We can see that this region is always in the direction of the deciding agent's goal. Figure 1-5c corresponds to the rule R_2. Here, the blue region denotes the region where the filter condition is satisfied. The gradient in the blue color relates to the scores computed by the map function (darker colors mean higher scores)–i.e. the rule chooses the agent with the darkest value.

The simple discrete structure in the rules and the visualizations of the learned linear functions allow us to interpret the neurosymbolic transformers' attention component. For instance, in the above example, we can interpret that the agents attend to their closest agent in the same group to avoid collisions and to an agent in the other group in their direction of travel for long-term planning.

In addition to achieving interpretability, we show that we can additionally optimize for combinatorial objectives (i.e. minimize the number of communications) when training a neurosymbolic transformer. Figure 1-4c shows the attention graph obtained using a neurosymbolic transformer, and we can see that it is sparser than the corresponding graph obtained with a DNN transformer in Figure 1-4b. Moreover, we show, in Section 7.7.4, that a neurosymbolic transformer is more robust and generalizable than a DNN transformer model to different distributions of agents (that is different from the training settings).

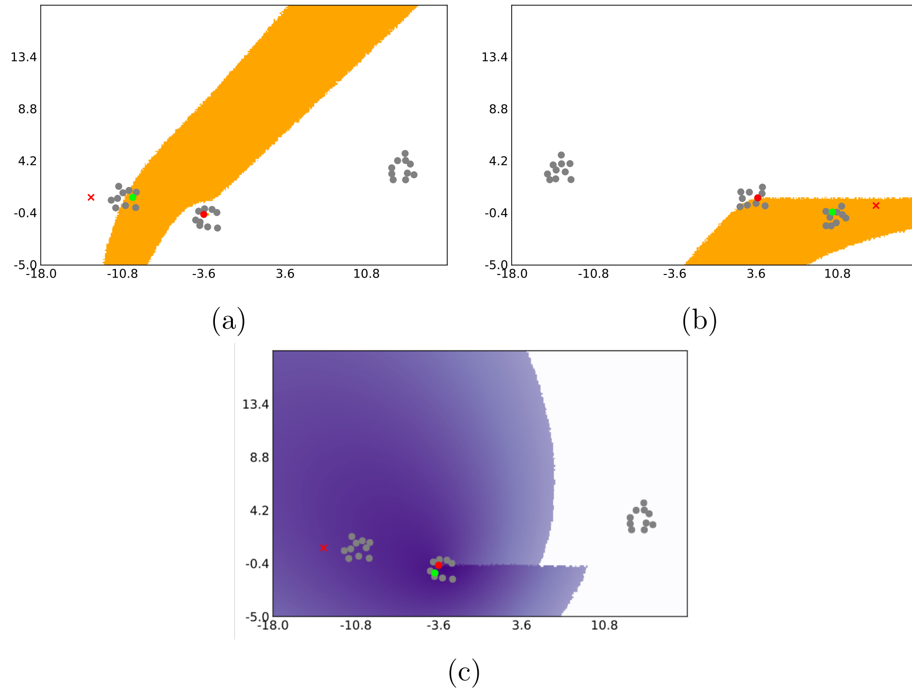


Figure 1-5: Visualization of the programmatic attention layer in Figure 1-4d, which has two rules R_1 and R_2. In this task, there are three groups of agents. The red circle denotes the agent currently choosing an action, the red cross denotes its goal, and the green circle denotes the agent selected by the rule. (a,b) Visualization of rule R_1 for two different states; orange denotes the region where the filter condition is satisfied—i.e., R_1 chooses a random agent in this region. (c) Visualization of rule R_2, showing the scores output by the map operator; darker values are higher—i.e., the rule chooses the agent with the darkest value.

Chapter 2

Related Works

Before elaborating on our approaches, we will look at some related works and explain how this thesis differs.

2.1 Program Synthesis

Program synthesis is a field focused on automatically generating programs based on specifications. Let us say we want to do a task like sorting a list. Instead of writing a program to do this task manually, the idea of program synthesis is to use a synthesizer and give it a high-level specification for the task. This specification can be input-output examples or some desired properties about the output. Then, we provide the synthesizer with a list of program components, e.g., for this sorting task, we might know that plausible a program will have loops, branches, and some functions like swap and peek. The job of the synthesizer is to find a program using these components that satisfies the specification.

The program synthesis idea has gained much popularity in the last 15 years as a technique to help software developers write code. The community can now synthesize programs for data structure manipulations [90, 71, 104, 75], can generate SQL queries [102], can generate programs to do text manipulation [40, 76] (a famous example for this is flash fill [40] which is a tool shipped with Microsoft excel), and so on.

There are a wide variety of synthesis techniques developed for the different domains such as constraint-based [92], enumerative [96], version space algebra [40] and stochastic [82]. However, most of these techniques are focused on synthesizing purely discrete programs.

Our work, in this thesis, aims to learn programmatic models for AI problems (such as learning control policies) rather than just software coding problems. As a consequence, these program models require both discrete and continuous components. These programs, in turn, require new learning algorithms that combine the above program synthesis techniques with deep learning techniques.

2.2 Intersection of Program Synthesis and Machine Learning

There has been a growing number of works in the intersection of program synthesis and machine learning [57, 29, 30, 28, 97, 105, 31, 69, 68, 100, 99, 10, 107, 86]. To compare these works with this thesis, it is helpful to distinguish them with respect to (1) the kinds of symbolic or neurosymbolic models they learn, (2) their learning algorithms, and (3) their target applications.

Symbolic/Neurosymbolic Model Classes

In many of the above works, the final models are purely discrete programs (from user-defined domain-specific languages), but they use deep neural network intermediates to help learn the programs [29, 28, 68, 86]. In some works, the model is a deep neural network, but the output of the neural network is a program [30, 31, 69]. There is another related work that learns probabilistic programs for classification tasks where the probabilistic semantics are used to capture the noise in the task [57].

The model classes explored in this thesis are most closely related to these works [100, 10, 107, 99] in that they have both discrete and continuous components. In particular, [10] learns decision trees and [100, 99, 107] learn programs with conditionals; the

programs in [100, 99] additionally can contain high-level list-based operators such as map and fold (similar to our attention programs in the neurosymbolic transformers).

The works in [97] and [105] use a combination of neural networks and programs. In particular, [97] learns programs to compose a library of neural functions. [105] first learns a program with loops to represent the high-level structural information of an image generation process, then they use a neural network to complete the image obtained after rendering the synthesized program.

This thesis adds to these different neurosymbolic models by introducing models with internal memory (state machine models) and neurosymbolic versions of transformers.

Learning Algorithms

Concerning the learning algorithms explored by these prior works in the intersection of program synthesis and machine learning, several works have explored the imitation learning algorithm (i.e. first training a DNN and then using it as an oracle to learn the neurosymbolic model) [100, 107]. [10] uses a more sophisticated version of imitation learning based on the Dagger algorithm [80] and additionally, uses the learned Q-function to weigh the data-points from the oracle towards more critical data-points. In this thesis, we too use a simple imitation learning algorithm to train neurosymbolic transformers.

Similar to how we designed the adaptive teaching algorithm to overcome the limitations in imitation learning, [99] devised another technique called imitation-projected gradient descent. This approach uses a form of mirror descent that takes a gradient step into the unconstrained DNN space and then projects back onto the constrained programmatic space. However, it is not clear if this technique can be applied to learning state machines since it is hard to get supervision on the internal memory of the state machine using a DNN oracle.

The learning approaches used in the other works are orthogonal to ours because their primary goal, unlike ours, is to use neural networks to help program synthesis problems. For instance, [31, 69, 30] learn DNNs that take in input-output examples

and produce programs satisfying these examples. The key idea in these works is to synthetically generate the supervised data for training the DNN by sampling programs from the DSL and generating fake input-output examples. [86] uses DNNs to relax a combinatorial program search space and use the performance of the learned DNN to prune out infeasible partial programs. [28] uses a DNN to learn a straight-chain program and then synthesizes a program with loops using traditional synthesis techniques. Similarly, [27] solves inverse graphics problems using geometric analysis techniques to find the continuous primitives and then using program synthesis techniques to figure out how to compose the primitives. Although not directly applicable in our settings, these techniques are still relevant because our algorithms internally have program synthesis components. Thus, the above approaches can make the program synthesis part of our algorithm faster and more efficient.

In addition to the above learning approaches, [97] and [105] employ different approaches to learn DNN+program models. Here, the main idea is to split the DNN training and the program synthesis parts into two separate independent steps with direct supervision for both components. However, there is no natural way to break the continuous learning and discrete learning for our problems; hence we require the teacher-student based algorithms.

Applications of Neurosymbolic Models

Neurosymbolic learning approaches, so far, have been applied in a number of different areas—such as program synthesis applications (list processing and text-editing tasks) [30, 31], visual domain (classifying hand-written characters [57], learning visual concepts [29], graphics problems [28, 27], generative modeling [105]), language domain (learning morphological rules [29], and instruction learning [69]). This thesis explores reinforcement learning tasks similar to [100, 99, 10, 107].

Similar to our work, these prior works have shown that symbolic/neurosymbolic models are better than deep neural networks in terms of achieving interpretability [10, 100, 28], few-shot learning [57, 29], safety [10, 107], and lifelong learning (ability to reuse neural network components learned in the previous tasks) [97].

2.3 Hybrid Systems

The hybrid systems literature [39] is relevant here because it is one of the earliest pieces of literature to explore the combination of discrete and continuous components. This community started by first developing techniques to verify that a manually written hybrid-model of a dynamical system achieves some desirable properties [4]. Then, the field gradually moved into synthesizing hybrid models for system identification problems [73]. However, the synthesis techniques developed there are limited; for instance, some works only focus on synthesizing the transition conditions while assuming that the dynamics in each mode is given [25], others use special algorithms (such as fix-point computation) that only work under certain linear assumptions [49].

Our approach (especially the state machine based models) builds upon this previously defined class of hybrid models. However, we view it as a generic model class for learning intelligent systems and present a new efficient learning technique that combines the latest techniques in machine learning and program synthesis.

2.4 Teacher-Student Algorithms in Other Settings

The idea of using a teacher to guide a student is seen in several other settings. For example, guided policy search [59] is a technique to train DNN policies for reinforcement learning tasks; it uses a teacher in the form of a trajectory optimizer to train a neural network student. Additionally, it has recently been shown that over-parameterization is essential in helping neural networks avoid local minima [2]. In our approach, the teacher can be seen as an over-parametric version of the student. Relaxing optimization problems by adding more parameters is a well established technique; in many cases, re-parameterization can make difficult non-convex problems solve efficiently [17, 16]. In the multi-agent RL community, there has also been a great deal of interest in using an oracle (e.g. a centralized policy) to train the final policy (e.g. a decentralized policy) [95]. The above works inspire our learning algorithms, but we adapted them to our setting where the student is a neurosymbolic model, and the

teacher is a continuous over-approximation (such as a DNN transformer).

2.5 Interpreting/Analyzing Neural Networks

One of the main contributions of this thesis is showing the interpretability of the learned neurosymbolic models. In the deep learning community, there are recent approaches to interpret and explain a learned neural network. For instance, prior works [70, 32, 64, 11] interpret a convolutional neural network by visualizing the features learned in each layer, which is in turn done by finding inputs that maximize activations of the given neurons. Some works try to explain individual decisions made by a neural network; [106] finds minimal symbolic corrections in the input to flip the current neural network's decision and uses these corrections to explain the neural network's decision. In contrast, the neurosymbolic models are naturally interpretable (without any need for additional optimizations to find the explanations). Moreover, they can be modified easily by an expert to get different behaviors.

2.6 Meta-Learning

Our generalization goal is related to that of meta-learning [34]; however, whereas meta-learning trains on a few examples from the novel environment, our goal is to generalize without additional training.

Chapter 3

Neurosymbolic Models

In this chapter, we introduce two kinds of neurosymbolic model classes: (i) *state machines*, which are suitable in place of RNNs and LSTMs because of their ability to store memory internally in the models, and (ii) *neurosymbolic transformers*, which are suitable to replace transformers by their ability to operate over lists of elements of arbitrary size.

3.1 State Machines

LSTMs and RNNs are capable of processing a sequence of inputs, learning to store some memory about the data seen so far, and using the memory and the input to produce outputs. These models are popularly used for speech recognition and sequence to sequence translation in natural language domains. They are also commonly used for learning reinforcement learning policies in the presence of partial observations [44].

At some level of abstraction, a RNN or a LSTM model π takes in the current input x_n and the current memory s_n, and produces an output y_n and the next memory s_{n+1}; $\pi(x_n, s_n) = (y_n, s_{n+1})$ where s_0 is initialized by some mechanism. In LSTMs and RNNs, this function π is usually modelled using complex deep neural networks. While they have been successfully used in many instances, some main drawbacks are the lack of interpretability of the learned model and the lack of generalization beyond the training distribution.

As an alternative, we introduce neurosymbolic state machines that are more interpretable and generalizable. Then, in Chapter 5, we describe the learning algorithm to train these state machines from data efficiently. At a high level, state machines are compositions of much simpler functions (e.g. linear functions). The internal memory of the state machine model (called its mode) indicates which simple function is currently used to generate the output. Thus, the state machine models are capable of encoding complex nonlinear functions such as iteratively repeating a sequence of simple functions (e.g., the logic needed for the car example in Figure 1-1). At the same time, state machines are substantially more structured than more typical model classes such as neural networks and decision trees.

3.1.1 Formalism

Let $x \in \mathcal{X}$ be an input to the model and $y \in \mathcal{Y}$ be the output and let's assume that the model is run on a sequence of inputs $x_0, \cdots, x_N$.

A state machine π is a tuple $\langle \mathcal{M}, \mathcal{H}, \mathcal{G}, m_s, m_e \rangle$. The *modes* $m_i \in \mathcal{M}$ of π are the internal memory of the state machine. Each mode $m_i \in \mathcal{M}$ corresponds to a *simple function* $H_{m_i} \in \mathcal{H}$, which is a function $H_{m_i} : \mathcal{X} \to \mathcal{Y}$ mapping the input to the output. When in mode m_i, the model outputs $y_n = H_{m_i}(x_n)$. Furthermore, each pair of modes (m_i, m_j) corresponds to a *switching condition* $G_{m_i}^{m_j} \in \mathcal{G}$, which is a function $G_{m_i}^{m_j} : \mathcal{X} \to \mathbb{R}$. When in mode m_i, if the model observes an input x_n such that $G_{m_i}^{m_j}(x_n) \geq 0$, then the model transitions from mode m_i to mode m_j. If there are multiple modes m_j with non-negative switching weight $G_{m_i}^{m_j}(x_n) \geq 0$, then the model transitions to the one that is greatest in magnitude; if there are several modes of equal weight, the model takes the first one according to a fixed ordering. Finally, $m_s, m_e \in \mathcal{M}$ are the start and the end modes, respectively; the state machine mode is initialized to m_s, and the state machine terminates when it transitions to m_e.

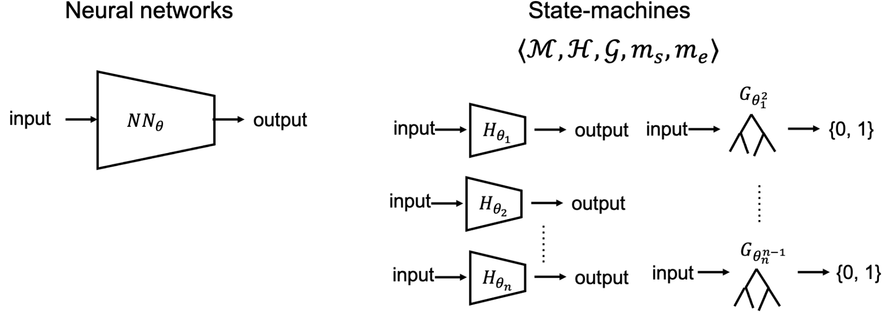


Figure 3-1: Neural networks vs state machine models — a conceptual comparison.

Formally, $\pi(x_n, s_n) = (y_n, s_{n+1})$, where $y_n = H_{s_n}(x_n)$, $s_0 = m_s$ and

$$s_{n+1} = \begin{cases} m^* = \mathrm{d}\mathsf{argmax}_m G^m_{s_n}(x_n) & \text{if } G^{m^*}_{s_n}(x_n) \geq 0 \\ s_n & \text{otherwise} \end{cases} \tag{3.1}$$

where d**argmax** is a deterministic **argmax** that breaks ties as described above.

The space of simple functions in the modes and the switching conditions are specified by *grammars* that encode the space of possible functions as a space of programs. Different grammars can be used for different problems. Typical grammars for the simple functions include scalar constants $\{C_\alpha : x \mapsto \alpha\}$ and linear functions $\{P^i_{\alpha,\alpha_0} : x \mapsto \alpha^\top.x + \alpha_0\}$. A typical grammar for switching conditions is the grammar

$$B \ ::= \ \{x[i] \leq \alpha\}_i \mid \{x[i] \geq \alpha\}_i \mid B_1 \wedge B_2 \mid B_1 \vee B_2$$

of Boolean predicates over the current input x, where $x[i]$ is the ith feature of x. In all these grammars, $\alpha_i \in \mathbb{R}$ are parameters to be learned. The grammar for switching conditions also has discrete parameters encoding the choice of expression.

For an example, see Figure 1-3, which is a learned state machine policy for the task in Figure 1-1. This state machine has three modes in addition to the start and the end modes. Note that our algorithm learns which edges connect which modes, i.e. the edges are not pre-specified in the grammar by a user. The grammar for the simple functions in each mode are scalar constants (one for each action output), and the grammar for the switching conditions are inequalities over some feature space x.

Figure 3-1 conceptually visualizes how a DNN model class differs from a state machine model class. Compared to a neural network model, which is just a huge function that takes in the current input and produces an output, a state machine model has n different functions for the n different modes. These functions are much simpler than a neural network. Then there are switching conditions (which is some subset of the n^2 edges)—each switching condition is a small decision tree that takes in the current input and produces true or false.

An interesting property about the space of state machines is that they can capture complex logic by increasing the number of modes. This property, in turn, allows us to control the complexity by restricting the maximum number of modes. So, it gives us a way to induce bias towards learning models with fewer modes, and simpler models usually are more interpretable and generalize better.

Note that, while we call the above class of models state machines, they are, in fact, built upon a particular subclass of state machines called hybrid automaton [45].

3.1.2 Discussion

Despite the several benefits obtained with state machines, there are a few limitations. A primary limitation of the state machines model class, in terms of expressiveness, is that they can only encode a single integer in its internal memory. Although this single integer of memory can still sufficiently express policies for many tasks, as seen in Chapter 6, it cannot encode all possible problems that RNNs and LSTMs can solve. Future directions involve looking into neurosymbolic models that combine state machines and other data structures such as stack and map to store more information internally. Another limitation of state machines is that currently, our learning algorithm only supports scalars/linear components in each mode. However, simple scalars and linear functions will not handle high dimensional inputs such as image inputs. In the future, we can explore more complex grammars for the functions in the modes and the switching conditions, for example, with some parts being small neural networks while still retaining the ability to learn generalizable behaviors.

3.2 Neurosymbolic Transformers

Like an LSTM, a transformer is a neural architecture for encoding one sequence into another one. A transformer relies upon the attention-mechanism, which looks at an input sequence and decides at each step which other parts of the sequence are important to compute the outputs. Recently, transformers gained immense popularity since they can learn long-distance relationships between the various elements of the input sequence. They are widely used in natural language processing (NLP) [98], vision [26], neural programmers [101], and to represent multi-agent policies in RL [22]. However, similar to other deep neural architectures, transformers lack interpretability and are difficult to generalize beyond the training distribution. Moreover, it is hard to enforce combinatorial objectives on the attention matrix, such as the sparsity constraints.

Our solution to handle these issues is a new class of models called neurosymbolic transformers. These are similar to transformers, but the attention networks are now programs from a domain-specific language (DSL). Below, we first formally define the transformer architecture and then extend it to present our neurosymbolic transformer architecture.

3.2.1 Transformers

The backbone of a neurosymbolic transformer is a plain-vanilla transformer model. A transformer maps an input sequence of N elements $x = (x_1, \cdots, x_N)$ to an output sequence of continuous representations $y = (y_1, \cdots, y_N)$. Each input element is encoded with its feature vector $x_i \in \mathcal{X} \in \mathbb{R}^{d_X}$ and each output element is continuous vector $y_i \in \mathcal{Y} \in \mathbb{R}^{d_Y}$. In contrast to the original transformer paper, our formalism also assumes the existence of directed relational features between any two input elements i.e. $r^{i,j} \in \mathcal{R} \in \mathbb{R}^{d_R}$ encodes the relational feature of the element x_j with respect to the element x_i.

A simple transformer model has one attention layer combined with some fully connected layers to produce the final desired outputs. An attention layer can be

described as mapping a query and a set of key-value pairs to an intermediate output. The queries, keys, values, and intermediate outputs are all vectors. This intermediate output is computed as a weighted sum of the values. The weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

Formally, a simple transformer model has four neural networks $\pi_\theta = (\pi_\theta^K, \pi_\theta^Q, \pi_\theta^V \pi_\theta^O)$. At a high level, $\pi_\theta^V : \mathcal{X} \times \mathcal{R} \rightarrow \mathbb{R}^{d_V}$ is the *value network*; $v^{i,j} = \pi_\theta^V(x^i, r^{i,j})$ is the value that element i contributes to the element j. $\pi_\theta^K : \mathcal{X} \times \mathcal{R} \rightarrow \mathbb{R}^{d_K}$ is the *key network*; $k^{i,j} = \pi_\theta^K(x^i, r^{i,j})$ is the key corresponding to the value $v^{i,j}$. $\pi_\theta^Q : \mathcal{X} \rightarrow \mathbb{R}^{d_K}$ is the *query network*; $q^i = \pi_\theta^Q(x^i)$ is the query proposed by the element i. $\pi_\theta^O : \mathbb{R}^{d_V + d_X} \rightarrow \mathbb{R}^{d_Y}$ is the final output network.

The attention layer computes a soft attention score $\alpha^{j,i}$ that indicates how much weight the element i places on the value $v^{j,i}$ that the element j contributes to the element i's query. The soft attention score is computed as

$$(\alpha^{1,i}, ..., \alpha^{N,i}) = \text{softmax}\left(\frac{\langle q^i, k^{i,1}\rangle}{\sqrt{d_K}}, ..., \frac{\langle q^i, k^{i,N}\rangle}{\sqrt{d_K}}\right). \tag{3.2}$$

The intermediate output of the attention layer i.e. the weighted sum of the values is then computed as

$$\lambda^i = \sum_{j=1}^{N} \alpha^{j,i} v^{j,i}. \tag{3.3}$$

and the final output of the transformer is the computed as

$$y^i = \pi_\theta^O(x^i, \lambda^i). \tag{3.4}$$

3.2.2 Attention Programs

In the formulation above, the key and the query networks compute the attention that an element x^i pays to the other elements x^j. An attention program aims to replace this computation using simple programmatic constructs.

We use the notation $[N] = \{1, \cdots, N\}$ to indicate the set of all integers from 1 to N. We use $r^i = \{r^{i,j}\}_j$ to indicate the list of relative features of all elements with respect to the element i. An attention program $P : \mathcal{X} \times \mathcal{R}^N \to [N]^K$ takes as input the feature vector of an element x^i and the set of relative features of all the other elements $r^i = \{r^{i,j}\}_j$, and computes a set of K other elements, $\mathcal{A}^i$, to attend to;

$$\mathcal{A}^i = P(x^i, r^i)$$

$j \in \mathcal{A}^i$ indicates the element i pays attention to the element j, otherwise it does not.

In our approach, an attention program is parameterized as a set of K rules $P = (R_1, ..., R_K)$ where each rule $R : \mathcal{X} \times \mathcal{R}^N \to [N]$ selects a single other element to attend to—i.e., $P(x^i, r^i) = (R_1(x^i, r^i), ..., R_K(x^i, r^i))$. There is a domain-specific language or grammar for the rules. Since these rules need to reason over a list of other elements, it is typical to have standard list operations such as map, fold, and filter in the grammar.

When a rule is applied to symbol i, it constructs a list

$$\ell = (s_1, ..., s_N) = ((x^i, r^{i,1}, 1), ..., (x^i, r^{i,N}, N)) \in \mathcal{S}^N,$$

where $\mathcal{S} = \mathcal{X} \times \mathcal{R} \times [N]$ encodes an cumulative feature vector of another agent. Then, the rule can apply the standard list operations to ℓ.

Some common operations are

- filter(B, ℓ): outputs the list of elements $s \in \ell$ such that $B(s) = 1$ where $B : \mathcal{S} \to \{0, 1\}$ is a Boolean predicate.

- map(F, ℓ) outputs the list of pairs $(F(s), j)$ for $s = (x^i, r^{i,j}, j) \in \ell$, where $F : \mathcal{S} \to \mathbb{R}$ is a scoring function.

- argmax inputs a list $((m^{j_1}, j_1), ..., (m^{j_H}, j_H)) \in (\mathbb{R} \times [N])^H$, where m^j is a score (or a metric) computed for element j, and outputs the element j with the highest metric m^j.

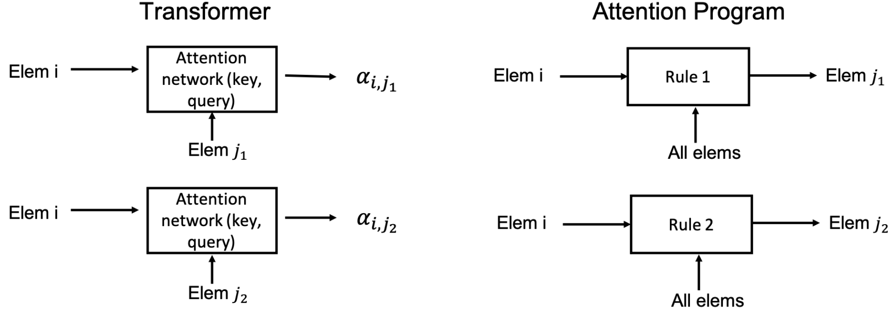


Figure 3-2: A traditional transformer's soft attention architecture vs an attention program.

- random(ℓ) takes as input a list $((x^i, r^{i,j_1}, j_1), ..., (x^i, r^{i,j_H}, j_H)) \in \mathcal{S}^H$, and outputs j_h for a uniformly random $h \in [H]$.

The grammar for the rules can include compositions of the above list operations. For example, argmax(map(F, filter(B, ℓ))) uses F to score every element after filtering based on the predicate B and then chooses the one with the best score. Another example, random(filter(B, ℓ)) randomly chooses one of the other symbols after filtering.

Finally, the filter predicates B and map functions F have their grammars; some typical grammars are:

$$B \ ::= \ \langle \theta, \phi(x^i, r^{i,j}) \rangle \geq 0 \mid B \wedge B \mid B \vee B \qquad\qquad F \ ::= \ \langle \theta, \phi(x^i, r^{i,j}) \rangle,$$

where $\theta \in \mathbb{R}^{d'}$ are weights, $\phi : \mathcal{X} \times \mathcal{R} \to \mathbb{R}^{d'}$ is a feature map, and $\langle x, y \rangle$ denotes dot product.

Figure 3-2 conceptually visualizes how a transformers attention network differs from an attention program. While a transformer computes a soft-attention for every pair of elements, each programmatic rule takes as input all the elements and chooses the single best one to attend to.

3.2.3 Combining Transformers with Attention Programs

A programmatic attention model P only chooses which elements to attend to; thus, we must combine it with the value network and the output network to get the final

output. These are combined by first using the outputs for P as masks to sparsify the soft attentions computed by the key and query networks. Note that an attention program P can be used to replace the key and the query networks, but, here, we only use P as a mask to obtain real number attention rather than Boolean values. The new attention values are computed as:

$$\alpha^{P,j,i} = \begin{cases} \alpha^{j,i}/Z & \text{if } j \in P(x^i, r^i) \\ 0 & \text{otherwise} \end{cases} \qquad \text{where} \qquad Z = \sum_{j \in P(x^i, r^i)} \alpha^{j,i} \tag{3.5}$$

where $\alpha^{j,i}$ are computed using Eq 3.2 and Z is the normalization constant.

Now, we can use $\alpha^{P,j,i}$ in place of $\alpha^{j,i}$ when computing the attention layer's intermediate output and the final output of the transformer.

$$\lambda^{P,i} = \sum_{j=1}^{N} \alpha^{P,j,i} v^{j,i} \tag{3.6}$$

$$y^{P,i} = \pi_\theta^O(x^i, \lambda^{P,i}) \tag{3.7}$$

3.2.4 Multiple Attention Layers

The above formulation can be extended to multiple attention layers. For the transformer architecture with two attention layers, first there is an *internal network* $\pi_\theta^H : \mathbb{R}^{d_X + d_V} \rightarrow \mathbb{R}^{d_H}$ that combines the input vector x^i and the cumulative value λ^i (eq 3.3) into an internal vector $h^i = \pi_\theta^H(x^i, \lambda^i)$. Next, we compute the next round of values as $v'^{i,j} = \pi_\theta^{V'}(h^i, r^{i,j})$ which replaces the input x^i in the original equation with the internal state h^i. New keys and queries are generated as $k'^{i,j} = \pi_\theta^{K'}(x^i, r^{i,j})$ and $q'^i = \pi_\theta^{Q'}(x^i)$, but these still use the original input x^i (to retain the interpretability of the inputs). Finally, the Equations 3.2, 3.3, and 3.4 are repeated for the second layer to compute the final output. This architecture can be extended similarly to an arbitrary number of attention layers. A neurosymbolic transformer with R layers will have R different programs (one for each layer).

3.2.5 Discussion

The expressivity of a neurosymbolic transformer is limited by the expressivity of the DSL for the attention program. In general, this issue is a common problem in any program synthesis system. The challenge is to find the right DSL that can capture the desired solutions and, at the same time, ensure that the DSL has a small search space to enable efficient search. Another limitation is that the attention programs can only operate on low-dimensional inputs (since the map functions and filter conditions contain simple linear functions). To handle high-dimensional inputs (e.g. images), we will need another neural network (e.g. an object segmentation network) to distil the abstract features for the program to operate on. Finally, in the current architecture, only the attention network is interpretable; the value network and the output network are still opaque neural networks.

Chapter 4

Neurosymbolic Reinforcement Learning

Although there are many potential applications for the neurosymbolic models presented so far, this thesis mainly focuses on learning neurosymbolic policies for reinforcement learning tasks. So at this point, it makes sense to introduce some background on reinforcement learning and why/how neurosymbolic policies are useful here.

4.1 Background on Reinforcement Learning

Reinforcement learning is a promising strategy for learning control policies for challenging sequential decision-making tasks. Recent work has demonstrated its promise in applications including game playing [67, 88], robotics control [20, 58], software systems [56, 19], and healthcare [78, 8]. A typical strategy is to build a high-fidelity simulator of the world, and then use reinforcement learning to train a control policy to act in this environment. This policy makes decisions (e.g., which direction to walk) based on the current state of the environment (e.g., the current image of the environment captured by a camera) to optimize the cumulative reward (e.g., how quickly the agent reaches its goal).

There has been significant recent progress on developing powerful *deep* reinforce-

ment learning algorithms [60, 85], which train a policy in the form of a deep neural network (DNN) by using gradient descent on the DNN parameters to optimize the cumulative reward. Importantly, these algorithms treat the underlying environment as a black box, making them very generally applicable.

However, most of the RL settings (such as autonomous cars) are safety-critical. Hence, having interpretable and verifiable policies is more crucial in RL settings than in other settings. Moreover, since we usually train the policies in simulation and test the policy in real-world settings, generalization is another key criterion. However, DNN policies are typically very difficult to understand and analyze, making it hard to guarantee their safety. The RL setting is particularly challenging since we need to reason not about isolated predictions but sequences of highly connected decisions.

As a consequence, it is interesting to learn policies in the form of neurosymbolic policies. They are significantly more interpretable than DNNs; consequently, human experts can often understand and debug behaviors of a neurosymbolic policy. In addition, in contrast to DNNs, neurosymbolic policies have discrete structure, making them much more amenable to formal verification, which can be used to prove correctness properties. Finally, neurosymbolic policies are more robust than their DNN counterparts—e.g., they generalize better to changes in the task or robot configuration.

4.2 Problem Formulation

We consider a reinforcement learning problem formulated as a *Markov decision process (MDP)* $M = (S, A, P, R)$ [77], where S is the set of states, A is the set of actions, $P(s' \mid a, s) \in [0, 1]$ is the probability of transitioning from state $s \in S$ to state $s' \in S$ upon taking action $a \in A$, and $R(s, a) \in \mathbb{R}$ is the reward accrued by taking action a in state s.

Given an MDP M, our goal is to train an agent that acts in M in a way that accrues high cumulative reward. We represent the agent as a policy $\pi : S \to A$ mapping states to actions. Then, starting from a state $s \in S$, the agent selects action

$a = \pi(s)$ according to the policy, observes a reward $R(s, a)$, transitions to the next state $s' \sim P(\cdot \mid s, a)$, and then iteratively continues this process starting from s'. For simplicity, we assume that a deterministic initial state $s_1 \in S$ along with a fixed, finite number of steps $H \in \mathbb{N}$. Then, we formalize the trajectory taken by the agent as a *rollout* $\zeta \in (S \times A \times \mathbb{R})^H$, which is a sequence of state-action-reward tuples $\zeta = ((s_1, a_1, r_1), ..., (s_H, a_H, r_H))$. We can sample a rollout by taking $r_t = R(s_t, a_t)$ and $s_{t+1} \sim P(\cdot \mid s_t, a_t)$ for each $t \in [H] = \{1, ..., H\}$; we let $D^{(\pi)}(\zeta)$ denote the distribution over rollouts induced by using policy π.

Now, our goal is to choose a policy $\pi \in \Pi$ in a given class of policies Π that maximizes the expected reward accrued. In particular, letting $J(\zeta) = \sum_{t=1}^{H} r_t$ be the cumulative reward of rollout ζ, our goal is to compute

$$\hat{\pi} = \operatorname*{arg\,max}_{\pi \in \Pi} J(\pi) \qquad \text{where} \qquad J(\pi) = \mathbb{E}_{\zeta \sim D^{(\pi)}}[J(\zeta)],$$

i.e., the policy $\pi \in \Pi$ that maximizes the expected cumulative reward over the induced distribution of rollouts $D^{(\pi)}(\zeta)$.

As an example, we can model a robot navigating a room to reach a goal as follows. The state $(x, y) \in S = \mathbb{R}^2$ represents the robot's position, and the action $(v, \phi) \in A = \mathbb{R}^2$ represents the robot's velocity v and direction ϕ. The transition probabilities are $P(s' \mid s, a) = \mathcal{N}(f(s, a), \Sigma)$, where

$$f((x, y), (v, \phi)) = (x + v \cdot \cos\phi \cdot \tau, y + v \cdot \sin\phi \cdot \tau),$$

where $\tau \in \mathbb{R}_{>0}$ is the time increment, and where $\Sigma \in \mathbb{R}^{2\times 2}$ is the variance in the state transitions due to stochastic perturbations. Finally, the rewards are the distance to the goal—i.e., $R(s, a) = -\|s - g\|_2 + \lambda \cdot \|a\|_2$, where $g \in \mathbb{R}^2$ is the goal and $\lambda \in \mathbb{R}_{>0}$ is a hyperparameter. Intuitively, the optimal policy $\hat{\pi}$ for this MDP takes actions in a way that maximizes the time the robot spends close to the goal g, while avoiding very large (and therefore costly) actions.

4.3 Neurosymbolic Policies

The main difference in neurosymbolic reinforcement learning compared to traditional reinforcement learning is the choice of policy class Π. In particular, we are interested in cases where Π is a space of neurosymoblic models of some form. In this section, we describe how/where state machines and neurosymbolic transformers can be used as policies.

4.3.1 State Machines for Tracking Internal State

A state machine policy is a good choice to represent a policy with internal state. In principle, for an MDP, keeping internal state is not necessary since the state variable contains all information necessary to act optimally. Nevertheless, in many cases, it can be helpful for the policy to keep internal state—for instance, for motions such as walking or swimming that repeat iteratively, it can be helpful to keep track of progress within the current iteration internally. In addition, if the state is partially observed (i.e., the policy only has access to $o = h(s)$ instead of the full state s), then internal state may be necessary to act optimally [50].

4.3.2 Neurosymbolic Transformers for List of States

In contrast to state machine policies, neurosymbolic transformers are designed to handle situations where the state of an agent includes a list of elements. For example, in multi-agent systems, the full state consists of a list of states for each individual agent [48]. In this case, the policy must compute a single action based on the given list of states. Alternatively, for environments with variable numbers of objects, the set of object positions must be encoded as a list. Finally, they can also be used to choose actions based on the history of the previous k states [100].

Chapter 5

Neurosymbolic Learning Algorithms

A key challenge with learning neurosymbolic models is that we cannot apply state-of-the-art machine learning algorithms. In particular, these algorithms are based on the principle of gradient descent on the model parameters. Yet, neurosymbolic policies are typically non-differentiable (or at least, their optimization landscape contains many local minima).

This problem is particularly challenging in reinforcement learning settings since the policy (the model) needs to output sequences of highly connected decisions. Moreover, the loss function (the reward) is typically very sparse, leading to numerous local minima. This chapter will focus on neurosymoblic learning algorithms for reinforcement learning settings.

In RL problems, the model is a policy $\pi : S \to A$ mapping states to actions. Then, starting from a state $s \in S$, the agent that is executing the policy selects action $a = \pi(s)$, observes a reward $R(s, a)$, transitions to the next state $s' \sim P(\cdot \mid s, a)$ according to a transition function P, and then iteratively continues this process starting from s'.

For continuous state and action spaces, state-of-the-art deep reinforcement learning algorithms [60, 85] consider a parameteric policy class $\Pi = \{\pi_\theta \mid \theta \in \Theta\}$, where the parameters $\Theta \subseteq \mathbb{R}^d$ are real-valued—e.g., π_θ is a DNN and θ are its parameters. Then, they compute π^* by optimizing over θ. One strategy is to use gradient descent

on the objective—i.e.,

$$\theta' \leftarrow \theta + \eta \cdot \nabla_\theta J(\pi_\theta).$$

In particular, the policy gradient theorem [93] encodes how to compute an unbiased estimator of this objective in terms of $\nabla_\theta \pi_\theta$. In general, most state-of-the-art approaches rely on gradient descent on the policy parameters θ. However, such approaches cannot be applied to training neurosymbolic policies since the search space of programs is typically discrete.

The same is also the case with sequence to sequence translations. There are good gradient-based approaches for DNNs, but they do not apply to learning neurosymbolic models.

Consequently, a common strategy for learning neurosymbolic models is to learn an intermediate teacher/oracle model (typically a deep neural model learned) and then use this intermediate oracle model to break down the original neurosymbolic learning problem into smaller problems. For example, we could convert the problem of learning a state machine policy for an RL problem into learning each individual small policy in the modes and the switching conditions separately with supervised data obtained using the oracle. The search landscape of these smaller neurosymbolic learning problems contains much fewer continuous parameters and/or fewer local optima. Therefore, they are amenable to either traditional program synthesis methods (enumeration/mcmc search) or traditional machine learning methods (decision tree learning or a simple gradient descent with multiple random initializations).

5.1 Imitation Learning

Training an oracle model first and then training a different model using supervision from the oracle model has been widely explored before as imitation learning. At a high level, the idea of imitation learning is first to use deep reinforcement learning to learn a high-performing DNN policy π^*, and then train the symbolic policy $\hat{\pi}$ to

imitate π^*.

A naïve strategy is to use an imitation learning algorithm called behavioral cloning [5], which uses π^* to explore the MDP, collects state-action pairs $Z = \{(s, a)\}$ pairs occurring in rollouts $\zeta \sim D^{(\pi^*)}$, and then trains $\hat{\pi}$ using supervised learning on the dataset Z—i.e.,

$$\hat{\pi} = \arg\min_{\pi \in \Pi} \sum_{(s,a) \in Z} \mathbb{1}(\pi(s) = a). \quad (5.1)$$

This simple imitation learning approach is a good fit for neurosymbolic transformers. However, it is not a great fit for state machine models because the oracle does not provide supervision on the state machine's internal memory. In Section 5.4, we describe another strategy to deal with this issue.

5.2 Imitation Learning for Neurosymbolic Transformers

For neurosymbolic transformers, the obvious choice for an oracle model is a normal transformer where the key, query, value, and output networks are all DNNs. Thus, the oracle is a neural network model $\pi_\theta = (\pi_\theta^K, \pi_\theta^Q, \pi_\theta^V, \pi_\theta^O)$ based on the transformer architecture [98] (Section 3.2.1); its parameters θ are trained using reinforcement learning to optimize $J(\pi_\theta)$.

Given an oracle model π_θ, we can precompute a dataset of tuples $Z = \{x, \alpha, y\}$ by sampling inputs, outputs and the intermediate attention values. Given a tuple in Z and a candidate attention program P, we can easily compute the corresponding values (α^P, y^P) for the same input x using the neurosymbolic transformer using Equations 3.5 and 3.7. Let Z^P be the set of tuples $(x, \alpha, y, \alpha^P, y^P)$.

Now a straight-forward imitation learning approach is to optimize P so that α

and α^P are similar for all the tuples in Z^P i.e.

$$\tilde{J}^{\alpha}(P;\theta) = -\mathbb{E}_{(x,\alpha,y,\alpha^P,y^P)\in Z^P}\left[\|\alpha - \alpha^P\|_1\right] \tag{5.2}$$

which can be optimized using traditional synthesis algorithms depending on the grammar of P. A drawback of the above objective is that we actually care about imitating the oracle's final output at the end of the day. A similar attention value does not necessarily mean a similar final output. So, a better objective is to optimize over P such that the final outputs of the two models are similar.

$$\tilde{J}^{y}(P;\theta) = -\mathbb{E}_{(x,\alpha,y,\alpha^P,y^P)\in Z^P}\left[\|y - y^P\|_1\right] \tag{5.3}$$

This objective needs to execute the output layer of the transformer π_θ^O for all x in the dataset Z and for every P considered (to compute y^P). Thus, this objective is computationally more expensive than the previous one but leads to a better-imitated model.

Re-training the Transformer. Once we synthesized a program P, we can form the combined neurosymbolic transformer policy $\pi_{P,\theta}$. One remaining issue is that the parameters θ are optimized for using the original soft attention weights $\alpha^{j,i}$ rather than the hard attention weights $\alpha^{P,j,i}$. Thus, we re-train the parameters of the transformer models in $\pi_{P,\theta}$. This training is identical to how π_θ was originally trained, except we use $\alpha^{P,j,i}$ instead of $\alpha^{j,i}$ to compute the output.

Extending to multiple attention layers. A neurosymbolic transformer with R attention layers will have R different programs (one for each layer). We can synthesize these programs independently. To synthesize the attention program P_r for the r-th layer, we use the hard attention weights α^{P_r} for the r-th layer and use the original soft attention weights for the other rounds $r' \neq r$ to compute the synthesis objective $\tilde{J}^y(P_r;\theta)$.

5.2.1 Optimizing Combinatorial Objectives

Another advantage of neurosymbolic transformers is the ability to include combinatorial objectives (especially on the attention graph structure). This is possible because we are already using combinatorial synthesis algorithms to handle the discreteness in the program grammar, and these algorithms can naturally handle combinatorial objectives. So, here, we can jointly optimize for the two objectives as:

$$\tilde{J}(P;\theta) = \tilde{J}^{y}(P;\theta) + \tilde{\lambda}\tilde{J}^{\alpha}$$

where $\tilde{\lambda} \in \mathbb{R}_{>0}$ is a hyperparameter, and the surrogate objective $\tilde{J}^{\alpha}$ aims to minimize some combinatorial objective on the attention graph α.

One useful kind of regularization objective here is to minimize the maximum degree of the attention graph (both incoming and outgoing).

$$\tilde{J}^{\alpha} = -\left(\max_{i} \sum_{j} \mathbb{1}(\alpha^{j,i} > 0) + \max_{i} \sum_{j} \mathbb{1}(\alpha^{i,j} > 0)\right)$$

The first term in the above equation corresponds to the maximum incoming degree and the second term corresponds to the maximum outgoing degree. The overall objective is to minimize the sum of these two terms. This objective essentially enforces that the output decision for any element (a robot) is only based on fewer elements (small incoming degree). At the same time, any particular element can only influence the output of a few other elements (small outgoing degree). This regularization is useful for interpretability (a sparse graph is easy to interpret), robustness, and generalization. Note that this regularization is hard to achieve with gradient descent based techniques (especially for the outgoing direction).

5.2.2 Program Synthesis for Supervised Learning

Recall that imitation learning reduces the complex neurosymbolic reinforcement learning problem to a simpler supervised learning problem. Here, we discuss algorithms for solving this supervised learning problem. In general, this problem is an instance

of *programming by example* [40, 41], which is a special case of program synthesis [42] where the task is specified by a set of input-output examples. In our setting, the input-output examples are the input-output pairs in the dataset Z used to train the attention programmatic.

An added challenge in applying program synthesis in machine learning settings is that traditional programming by example algorithms are designed to compute a program that correctly fits *all* of the training examples. In contrast, in machine learning, there typically does not exist a single program that fits all of the training examples. Instead, we need to solve a *quantitative* synthesis problem where the goal is to minimize the number of errors on the training data.

One standard approach to solving such program synthesis problems is to enumerate over all possible programmatic policies $\pi \in \Pi$. Π is specified as a context-free grammar, and we can use the standard algorithms to enumerate programs in that grammar (typically up to a bounded depth) [6]. In addition, domain-specific techniques can be used to prune provably suboptimal portions of the search space to speed up enumeration [18]. For particularly large search spaces, an alternative strategy is to use a stochastic search algorithm that heuristically optimzes the objective; for example, Metropolis Hastings can be used to adaptively sample programs (e.g., with the unnormalized probability density function taken to be the objective value) [83].

For the attention programs, the search space has both discrete choices (to choose between the different combinations of list-processing operations) and continuous parameters (for the linear functions in the map functions and filter conditions). For this space, we found the stochastic search algorithm (Metropolis Hastings) more suitable.

5.2.3 Discussion

Here, we presented a straightforward learning algorithm for training neurosymbolic transformers. This simplistic approach is great for plugging in various other algorithms for either the teacher or the student. For example, currently, we use a model-based reinforcement learning algorithm for the teacher that backpropagates through time [24, 9]. This algorithm can be replaced by any of the other sophisticated algo-

rithms such as [62]. Similarly, for the student, there are different synthesis algorithms, including several neural guided synthesis algorithms [30, 31, 68]. Some of these neural guided synthesis algorithms can even learn the high-level DSLs (e.g. common combinations of list-manipulating operations) from simple basic components (e.g. map and filter) [30].

The simple imitation learning algorithm can sometimes suffer from distribution shifts (although not observed in our settings). But, luckily, there are already techniques to address these issues, such as the Dagger algorithm [80].

5.3 Shortcomings of Imitation Learning

A fundamental shortcoming of these imitation algorithms is that they do not adjust the oracle model π^* to account for the limitations on the capabilities of the symbolic model $\hat{\pi}$. An oracle model can solve the task in many different ways. A naive imitation learning would not force the oracle to learn something that a compact symbolic model can imitate. Moreover, in the state machines case, a DNN teacher does not provide any supervision on the internal memory of the student.

Therefore, this thesis presents another approach called *adaptive teaching*, where rather than choosing π^* to be a DNN, it is instead a model whose structure mirrors that of $\hat{\pi}$. Then, we can directly update π^* on each training iteration to reflect the structure of $\hat{\pi}$. We showed adaptive teaching to be an effective strategy for learning state machine policies.

5.4 Adaptive Teaching

In this section, we first describe the general idea of the adaptive teaching algorithm and derive it using variational inference. Then, the next section will present how to instantiate this algorithm concretely for learning state machines.

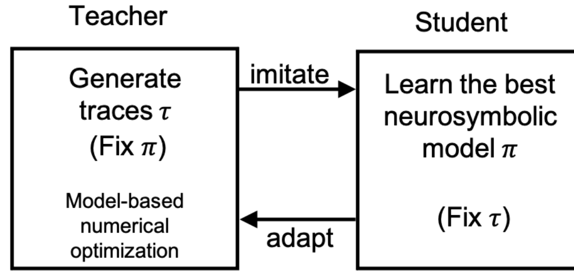


Figure 5-1: High-level overview of the adaptive teaching algorithm.

5.4.1 Overview

In the adaptive teaching algorithm, the teacher is abstractly represented as a collection of traces τ_x for each input x. An example of a trace for RL problems is a trajectory (a sequence of actions over a bounded horizon). A key insight is that we can parameterize τ_x in a way that mirrors the structure of the neurosymbolic model. For instance, in Section 5.5.1, we define a parameterization of τ_x that looks like a loop-free version of a state machine.

At a high level, the algorithm alternatively learns a teacher (a collection of τ_x for multiple initial inputs x) and a student (π). The goal for the teacher is two-fold (i) find traces that maximize the reward for the RL problem, and (ii) find traces that match the structure of the student learned so far. The latter objective is what distinguishes the adaptive teaching approach from imitation learning. The goal for the student is to "glue" the teacher's traces together using maximum likelihood to construct a neurosymbolic model. Figure 5-1 depicts this algorithm at a high level.

5.4.2 Adaptive Teaching via Variational Inference

Next, we derive the adaptive teaching formulation by reformulating the learning problem in the framework of probabilistic reinforcement learning, and also consider neurosymbolic models π that are probabilistic (see Section 5.5.3 for a probabilistic state machine model). Then, we use a variational approach to break the problem into the

teacher and the student steps. The log-likelihood of a policy π is defined as follows:

$$\ell(\pi) = \log \mathbb{E}_{p(\tau|\pi)}[e^{-\lambda L(\tau)}] \tag{5.4}$$

where $p(\tau \mid \pi)$ is the probability of getting the trace τ when using the model π, $\lambda \in \mathbb{R}_{\geq 0}$ is a hyperparameter, and $L(\tau)$ is the loss (negative reward) assigned to the trace τ. Now, we can use the variational approach to express Eq (5.4) as an expectation over an auxiliary distribution $q(\tau)$ which will define the teacher. We have

$$\ell(\pi) = \log \mathbb{E}_{q(\tau)}\left[e^{-\lambda L(\tau)} \cdot \frac{p(\tau \mid \pi)}{q(\tau)}\right] \geq \mathbb{E}_{q(\tau)}[-\lambda L(\tau) + \log p(\tau|\pi) - \log q(\tau)] \tag{5.5}$$

where $q(\tau)$ is the variational distribution and the inequality follows from Jensen's inequality. Thus, we can optimize π by maximizing the lower bound Eq (5.5) on $\ell(\pi)$. Since the first and third term of Eq (5.5) are constant with respect to π, we have

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{q(\tau)}[\log p(\tau|\pi)]. \tag{5.6}$$

Next, the optimal choice for q (i.e., to minimize the gap in the inequality in Eq (5.5)) is

$$q^* = \arg\min_{q} D_{\mathrm{KL}}(q(\tau) \parallel e^{-\lambda L(\tau)} \cdot p(\tau \mid \pi)/Z) \tag{5.7}$$

where Z is a normalizing constant. We choose q to have form $q(\tau) = p(x) \cdot \delta(\tau - \tau_x)$, where δ is the Dirac delta function, $p(x)$ is the input distribution, and τ_x are the parameters to be optimized, where τ_x encodes the trace for the input x. Then, up to constants, the objective of Eq (5.7) equals

$$\mathbb{E}_{p(x)}\left[\log p(x) + \mathbb{E}_{\delta(\tau-\tau_x)}[\log \delta(\tau - \tau_x)] - (-\lambda L(\tau_x) + \log p(\tau_x \mid \pi, x))\right].$$

The first term is constant; the second term is degenerate, but it is also constant. Thus, we have

$$q^* = \arg\max_{\{\tau_x\}} \mathbb{E}_{p(x)} \left[-\lambda L(\tau_x) + \log p(\tau_x \mid \pi, x) \right]. \tag{5.8}$$

Thus, we can optimize Eq (5.4) by alternatingly optimizing Eq (5.6) and Eq (5.8).

We interpret these equations as adaptive teaching. At a high level, the teacher (i.e., the variational distribution q^* in Eq (5.8)) is used to guide the optimization of the student (i.e., the neurosymbolic model π^* in Eq (5.6)). Rather than compute the teacher in closed form, we approximate it by sampling finitely many inputs $x^k \sim X$ and then computing the optimal trace for x^k. Formally, on the ith iteration, the teacher and student are updated as follows:

$$\textbf{Teacher} \quad q_i^* = \sum_{k=1}^{K} \delta(\tau_k^i) \tag{5.9}$$

$$\text{where} \quad \tau_k^i = \mathsf{argmax}_\tau \; -\lambda L(\tau) + \log p(\tau \mid \pi^{i-1}, x^k) \quad (x^k \sim X)$$

$$\textbf{Student} \quad \pi_i^* = \mathsf{argmax}_\pi \; \sum_{k=1}^{K} \log p(\tau_k^i \mid \pi, x^k) \tag{5.10}$$

The teacher objective Eq (5.9) is to both minimize the loss $L(\tau)$ for a sampled input x and to maximize the probability $p(\tau \mid \pi, x)$ of obtaining the trace τ for the input x according to the current student π. The latter encourages the teacher to match the structure of the student. Furthermore, the teacher is itself updated at each step to account for the changing structure of the student. The student objective Eq (5.10) is to imitate the distribution of traces according to the teacher.

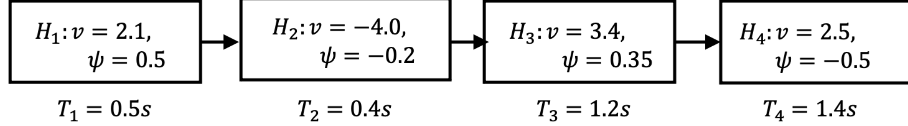


Figure 5-2: An example loop-free policy. This policy has a sequence of 4 simple policies. The grammar for simple policies, in this case, is scalar constants (same as the state machine policy in Figure 1-3). Each simple policy H_i is executed for T_i duration before switching to the next policy.

5.5 Instantiating Adaptive Teaching for Learning State Machine Policies

5.5.1 Trace Parameterization

One approach is to parameterize τ as an arbitrary action sequence $(a_0, a_1, ...)$ and use gradient-based optimization to compute τ. However, this approach can perform poorly—even though we regularize τ towards the student, it could exhibit behaviors that are hard for the student to capture. Instead, we parameterize τ in a way that mirrors the student. In particular, we parameterize τ like a state machine, but rather than having modes and switching conditions that adaptively determine the sequence of simple policies (in the modes) to be executed and the duration of execution, the sequence of simple policies is fixed and each simple policy is executed for a fixed duration.

More precisely, we represent τ as an *loop-free policy* $\tau = \langle \mathcal{H}, \mathcal{T} \rangle$. To execute τ, each simple policy $H_i \in \mathcal{H}$ is applied for the corresponding duration $T_i \in \mathcal{T}$, after which H_{i+1} is applied. The simple policies are from the same grammar of simple policies for the student. Figure 5-2 shows an example of a loop-free policy. The obvious way to represent a duration T_i is as a number of time steps $T_i \in \mathbb{N}$. However, with this choice, we cannot use continuous optimization to optimize T_i. Instead, we fix the number of discretization steps P for which H_i is executed, and vary the time increment $\Delta_i = T_i/P$—i.e., $x_{n+1} \approx x_n + F(x_n, H_i(o)) \cdot \Delta_i$. We enforce $\Delta_i \leq \Delta_{\max}$ for a small $\Delta_{\max}$ to ensure that the discrete-time approximation of the dynamics is sufficiently accurate.

5.5.2 Teacher's Optimization

We use numerical optimization to solve the trace optimization problem Eq (5.9)—i.e., computing τ_k for a given input x^k. The main challenge is handling the term $p(\tau \mid \pi, x)$ in the objective. Symbolically computing this probability is hard because of the discrete-continuous structure of π. Another alternative is to precompute the probabilities of all traces τ that can be derived from π. However, this is also infeasible because the number of traces is unbounded. Thus, we perform trace optimization in two phases. First, we use a sampling-based optimization algorithm to obtain a set of good traces $\tau^1, ..., \tau^L$ from π for each x. Then, we apply gradient-based optimization, replacing $p(\cdot \mid \pi, x)$ with a term that regularizes τ to be close to $\{\tau^\ell\}_{\ell=1}^L$.

The first phase proceeds as follows: (i) sample $\tau^1, \cdots, \tau^L$ using π for x, and let p^ℓ be the probability of τ^ℓ according to π, (ii) sort these samples in decreasing order of objective $p^\ell \cdot e^{-\lambda L(\tau^\ell)}$, and (iii) discard all but the top ρ samples. This phase essentially performs one iteration of CEM [65]. Then, in the second phase, we replace the probability expression with $p(\tau \mid \pi, x) \approx \frac{\sum_{\ell=1}^{\rho} p^\ell \cdot e^{-d(\tau, \tau^\ell)}}{\sum_{\ell=1}^{\rho} p^\ell}$, which we use gradient-based optimization to optimize. Here, $d(\tau, \tau^\ell)$ is a distance metric between two loop-free policies, defined as the L_2 distance between the parameters of τ and τ^ℓ. In our experiments in Chapter 6, we chose the number of samples, $\rho = 10$. For our benchmarks, we did not notice any improvement in the number of student-teacher iterations by increasing ρ above 10. So, we believe we are not losing any information from this approximation.

5.5.3 Student's Imitation Learning

Next, we describe how the student solves the maximum likelihood problem Eq (5.10) to compute π^*.

Probabilistic State Machines.

Although the output of our algorithm is a student policy that is a deterministic state machine, our algorithm internally relies on distributions over traces induced by the

student policy to guide the teacher. Thus, we represent the student policy as a probabilistic state machine during learning. To do so, we simply make the simple functions H_{m_j} and switching conditions $G_{m_{j_1}}^{m_{j_2}}$ probabilistic—instead of constant parameters in the grammar for simple functions and switching conditions, now we have Gaussian distributions $\mathcal{N}(\alpha, \sigma)$. Then, when executing π, we obtain i.i.d. samples of the parameters $H'_{m_j} \sim H_{m_j}$ and $\{(G_{m_j}^{m'_j})' \sim G_{m_j}^{m'_j}\}_{m'_j}$ every time we switch to mode m_j, and act according to H'_{m_j} and $\{(G_{m_j}^{m'_j})'\}$ until the mode switches again. By re-sampling these parameters on every mode switch, we avoid dependencies across different parts of a rollout or different rollouts. On the other hand, by not re-sampling these parameters within a mode switch, we ensure that the structure of π remains intact within a mode.

Optimization

Each τ_k can be decomposed into segments (k, i) where the simple function $H_{k,i}$ is executed for $T_{k,i}$ iterations. For example, each block in Figure 5-2 is a segment. Furthermore, for the student π, let H_{m_j} be the simple function distribution for mode m_j and $G_{m_{j_1}}^{m_{j_2}}$ be the switching condition distribution for mode m_{j_1} to mode m_{j_2}. Note that H_{m_j} and $G_{m_{j_1}}^{m_{j_2}}$ are distributions whereas $H_{k,i}$ and $T_{k,i}$ are constants. We have

$$p(\tau_k \mid \pi, x^k) = \prod_i p(H_{k,i} \mid \pi, x^k) \cdot p(T_{k,i} \mid \pi, x^k).$$

For each segment (k, i), let $\mu_{k,i}$ be the latent random variable indicating the mode used by π to generate the segment (k, i); in particular, $\mu_{k,i}$ is a categorical random variable that takes values in the modes $\{m_j\}_j$. And $\mu_{k,i} = m_j$ means that $H_{k,i}$ is sampled from the distribution H_{m_j} and $T_{k,i}$ is determined by the sampled switching conditions from distributions $\{G_{m_j}^{m'_j}\}_{j'}$. Assuming the presence of latent variable $\mu_{k,i}$, allows the student to compute π^* by computing $H_{m_j}^*$ and ${G_{m_{j_1}}^{m_{j_2}}}^*$ independently.

Since directly optimizing the maximum likelihood π is hard in the presence of the latent variables $\mu_{k,i}$, we use the standard expectation maximization (EM) approach to optimizing π, where the E-step computes the distributions $p(\mu_{k,i} = m_j)$ assuming

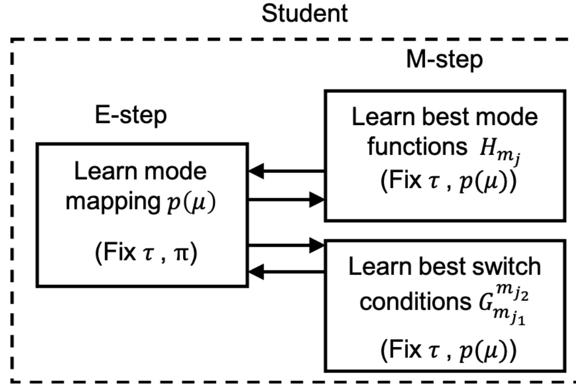


Figure 5-3: The high-level overview of the student's EM approach to imitate the teacher for learning state machine policies.

π is fixed, and the M-step optimizes π assuming the probabilities $p(\mu_{k,i} = m_j)$ are fixed. See Section 5.5.5 for more details about this EM approach. An overview of the student's imitation learning algorithm is shown in figure 5-3.

5.5.4 Example

Figure 5-4 shows two iterations of the adaptive teaching algorithm. Figure 5-4(a) and (d) show examples of loop-free policies for two different initial states and two different teacher iterations. These traces are visualized as a sequence of boxes. The colors signify different simple policy parameters (in Figure 5-2, a simple policy is parameterized as two scalars). The lengths of the boxes signify the duration of the segments. In Figure 5-4, (c) and (f) show the most probable traces from the state machine policies learned at the end of the EM approach for two different student iterations. In Figure 5-4, (b) and (e) show the learned mode mappings $p(\mu = m_j)$ for the segments in the loop-free policies shown in (a) and (d) respectively. Here, each box shows the composition of the modes vertically distributed according to their probabilities. Note how the loop-free policies in (d) are regularized to match the student's state machine policy learned in the previous iteration (c).

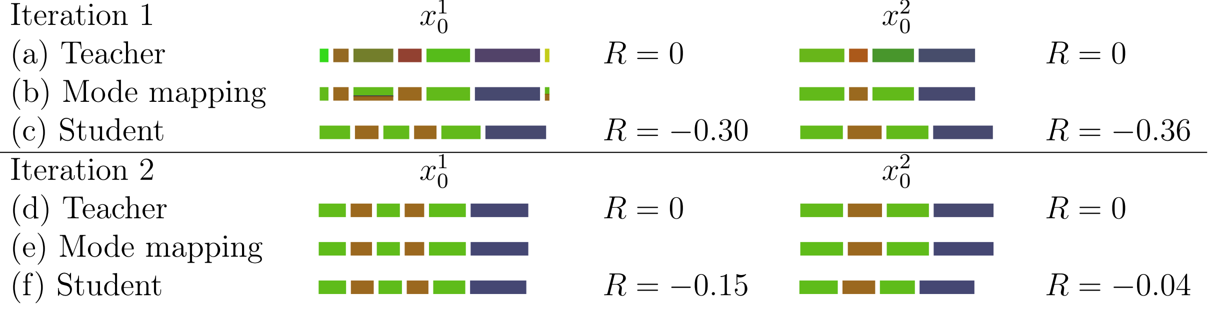


Figure 5-4: Visualization showing the student-teacher interaction for two iterations. (a) The loop-free policies (with their corresponding rewards) learned by the teacher for two different initial states. Here, the boxes signify the different segments in the loop-free policies, the colors signify different actions, and the boxes' lengths signify the segments' durations. (b) The mapping between the segments and the modes in the state machine—i.e., $p(\mu = m_j)$. Each box shows the composition of the modes vertically distributed according to their probabilities. For example, the third segment in the loop-free policy for x_0^1 has $p(\mu = \text{Green}) = 0.65$ and $p(\mu = \text{Brown}) = 0.35$. (c) The most probable rollouts from the state machine policy learned by the student. Finally, (d), (e) and (f) are similar to (a), (b) and (c), but for the second iteration.

5.5.5 EM Approach Details

First, note that we have

$$p(\tau_k \mid \pi, x^k) = \prod_i p(H_{k,i} \mid \pi, x^k) \cdot p(T_{k,i} \mid \pi, x^k).$$

where

$$p(H_{k,i} \mid \pi, x^k) = \sum_j p(H_{k,i} \mid H_{m_j}) \cdot p(\mu_{k,i} = m_j).$$

Similarly, the duration $T_{k,i}$ is determined both by the current mode $\mu_{i,k} = m_{j_1}$, and by the switching conditions $G^-_{m_{j_1}} = \{G^{m_{j_2}}_{m_{j_1}}\}_{m_{j_2}}$ from the current mode m_{j_1} into some other mode m_{j_2}. More precisely, let $\gamma_{k,i}$ denote the trajectory (sequence of states) of the (k,i) segment of τ_k, and let $\zeta(\gamma_{k,i}, G^-_{m_j})$ denote the earliest time at which a switching condition $G \in G^-_{m_j}$ becomes true along $\gamma_{k,i}$. Since $G \in G^-_{m_j}$ are

distributions, $\zeta(\gamma_{k,i}, G^-_{m_j})$ is a distribution on transition times. Then, we have

$$p(T_{k,i} \mid \pi, x^k) = \sum_{m_{j_1}} \sum_{m_{j_2}} p(\mu_{k,i} = m_{j_1}) \cdot p(\mu_{k,i+1} = m_{j_2}) \cdot p(T_{k,i} \mid G^{m_{j_2}}_{m_{j_1}}, G^-_{m_{j_1}})$$

$$p(T_{k,i} \mid G^{m_{j_2}}_{m_{j_1}}, G^-_{m_{j_1}}) = p(T_{k,i} = \zeta(\gamma_{k,i}, G^{m_{j_2}}_{m_{j_1}})) \cdot \prod_{m_{j_3} \neq m_{j_2}} p(T_{k,i} < \zeta(\gamma_{k,i}, G^{m_{j_3}}_{m_{j_1}})).$$

In other words, $T_{k,i}$ is the duration until $G^{m_{j_2}}_{m_{j_1}}$ triggers, conditioned on none of the conditions $G^{m_{j_3}}_{m_{j_1}}$ triggering (where $m_{j_3} \neq m_{j_2}$).

Numerically optimizing the maximum likelihood objective to compute π^* is hard because it requires integrating over all possible choices for the latent variables $\mu_{k,i}$. For example, if the teacher generates 10 loop-free policies every iteration and there are 10 modes in each loop-free policy, and 4 modes in the state machine, the number of choices for the latent variables is 4^{100}, which makes the enumeration infeasible. The expectation-maximization method provides an efficient way for computing the maximum likelihood, by alternatingly optimizing for the latent variables and the state machine parameters. The E-step computes the probability distributions $p(\mu_{k,i} = m_j)$ for a fixed π, and the M-step optimizes H_{m_j} and $G^{m_{j_2}}_{m_{j_1}}$ given $p(\mu_{k,i} = m_j)$.

E-step. Assuming π is fixed, we have

$$p(\mu_{k,i} = m_j \mid \pi, \{\tau_k\}) = \frac{p(H_{k,i} \mid H_{m_j}) \cdot p(T_{k,i} = \zeta(\gamma_{k,i}, G^-_{m_j}))}{\sum_{m'_j} p(H_{k,i} \mid H_{m'_j}) \cdot p(T_{k,i} = \zeta(\gamma_{k,i}, G^-_{m'_j}))}. \quad (5.11)$$

M-step. Assuming $p(\mu_{k,i} = m_j)$ is fixed, we solve

$$\mathsf{argmax}_{\{H_{m_j}\}} \sum_{k,i} p(\mu_{k,i} = m_j) \cdot \log p(H_{k,i} \mid H_{m_j}) \quad (5.12)$$

$$\mathsf{argmax}_{\{G^{m_{j_2}}_{m_{j_1}}\}} \sum_{k,i} p(\mu_{k,i} = m_{j_1}) \cdot p(\mu_{k,i+1} = m_{j_2}) \cdot \log p(T_{k,i} = \zeta(\gamma_{k,i}, G^{m_{j_2}}_{m_{j_1}}))$$
$$+ p(\mu_{k,i} = m_{j_1}) \cdot (1 - p(\mu_{k,i+1} = m_{j_2})) \cdot \log p(T_{k,i} < \zeta(\gamma_{k,i}, G^{m_{j_2}}_{m_{j_1}})) \quad (5.13)$$

or $G^{m_{j_2}}_{m_{j_1}}$, the first term handles the case $\mu_{k,i+1} = m_{j_2}$, where we maximize the probability that $G^{m_{j_2}}_{m_{j_1}}$ makes the transition at duration $T_{k,i}$, and the second term handles

the case $\mu_{k,i+1} \neq m_{j_2}$, where we maximize the probability that $G_{m_{j_1}}^{m_{j_2}}$ does not make the transition until after duration $T_{k,i}$.

We briefly discuss how to solve these equations. For the mode functions, suppose that H encodes the distribution $\mathcal{N}(\alpha_H, \sigma_H^2)$ over the simple function parameters. Then, we have

$$\alpha^*_{H_{m_j}} = \frac{\sum_{k,i} p(\mu_{k,i} = m_j) \cdot \alpha_{H_{k,i}}}{\sum_{k,i} p(\mu_{k,i} = m_j)} \tag{5.14}$$

$$(\sigma^*_{H_{m_j}})^2 = \frac{\sum_{k,i} p(\mu_{k,i} = m_j) \cdot (\alpha_{H_{k,i}} - \alpha^*_{H_{m_j}})(\alpha_{H_{k,i}} - \alpha^*_{H_{m_j}})^T}{\sum_{k,i} p(\mu_{k,i} = m_j)} \tag{5.15}$$

Solving for the parameters of $G_{m_{j_1}}^{m_{j_2}}$ is more challenging, since there can be multiple kinds of expressions in the grammar that are switching conditions, which correspond to discrete parameters, and we need to optimize over these discrete choices. To do so, we perform a greedy search over these discrete choices (see Section 5.5.6 for details on the greedy strategy). For each choice considered during the greedy search, we encode Eq (5.13) as a numerical optimization problem and solve it to compute the corresponding means $\alpha^*_{G_{m_{j_1}}^{m_{j_2}}}$ and standard deviations $\sigma^*_{G_{m_{j_1}}^{m_{j_2}}}$ Then, we choose the discrete choice that achieves the best objective value according to Eq (5.13).

Computing the optimal parameters for switching conditions is more expensive than doing so for the simple functions in the modes; thus, on each student iteration, we iteratively solve Eq (5.11) and Eq (5.12) multiple times, but only solve Eq (5.13) once.

The EM method does not guarantee global optima but usually works well in practice. In addition, since computing the switching conditions is expensive, we had to restrict the number of EM iterations. However, note that even if the EM algorithm didn't converge, our overall algorithm can still recover by using additional teacher-student interactions.

The alternate method would be to run the EM algorithm multiple times/longer to get better results per student iteration, and "maybe" reduce the total number of teacher-student iterations. We say "maybe" because the EM algorithm might have

already converged to the global optima, making the extra EM iterations useless. The trade-off between our approach and this alternative depends on whether the teacher's algorithm or the student's algorithm is expensive for a particular benchmark.

However, from Figure 6-16 in Chapter 6, we can see that some of our benchmarks already use very few (< 5) teacher-student iterations (Car, QuadPO, Pendulum, Mountain car, and Swimmer). Of the other three benchmarks that needed many iterations, for two of them (Cartpole and Acrobot), the student's algorithm is as expensive as the teacher's algorithm. This justifies our decision to not run the EM algorithm multiple times/longer.

5.5.6 Synthesizing Switching Conditions

Next, we describe how we search over the large number of discrete choices in the grammar for switching conditions. It is not hard to show that in Eq (5.13), the objectives for the switching condition parameters $G_{m_{j_1}}^{m_{j_2}}$ corresponding to different transitions (m_{j_1}, m_{j_2}) decompose into separate problems. Therefore, we can perform the search for each transition (m_{j_1}, m_{j_2}) separately. For each transition, the naïve approach would be to search over the possible derivations in the context-free grammar for switching conditions to some bounded depth. However, this search space is exponential in the depth due to the productions $B ::= B \wedge B$ and $B ::= B \vee B$. Thus, we employ a greedy search strategy to avoid the exponential blowup.

Intuitively, our search strategy is to represent switching conditions as a kind of decision tree, and then perform a greedy algorithm to search over decision tree[1]. Our search strategy is similar to (but simpler than) the one in [13]. In particular, we can equivalently represent a switching condition as a decision tree, where the internal nodes have the form $x[i] \leq \alpha$ or $x[i] \geq \alpha$ (where $i \in \{1, ..., d_x\}$ and $\alpha \in \mathbb{R}$ are parameters), and the leaf nodes are labeled with "Switch" or "Don't switch"—e.g., Figure 5-5 shows two examples of switching conditions expressed as decision trees. Then, our algorithm initializes the switching condition to a single leaf node—i.e., $G_{\text{cur}} \leftarrow$ "Switch". At each step, we consider switching conditions $G \in \text{next}(G_{\text{cur}})$ that

[1]However, our algorithm is not very similar to greedy decision tree learning algorithms.

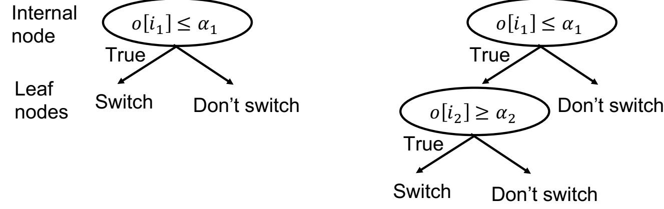


Figure 5-5: Switching conditions represented as decision trees.

Algorithm 1 Greedy algorithm for learning switching conditions.

---

procedure LEARNSWITCHINGCONDITION
  $G_{\text{cur}} \leftarrow$ "Switch"
  while $|G_{\text{cur}}| < N$ **do**
    $G_{\text{cur}} \leftarrow \arg\min_{G \in \text{next}(G_{\text{cur}})} \text{cost}(G)$
  return G_{cur}

---

expand a single leaf node of G_{cur}; among these, we choose G_{cur} to be the one that minimizes a loss $\text{cost}(G)$.

More precisely, to construct $\text{next}(G_{\text{cur}})$, we iterate over all leaf nodes $L \in \text{leaves}(G_{\text{cur}})$, and all expressions $E \in \mathcal{E}$, where

$$\mathcal{E} = \left\{ \text{if } x[i] \sim \alpha \text{ then "Switch" else "Don't Switch"} \;\middle|\; i \in \{1, ..., d_x\},\ \alpha \in \mathbb{R},\ \ \sim \in \{\geq, \leq\} \right\}$$

Here, $\sim \in \{\geq, \leq\}$ is a inequality relation, $i \in \{1, ..., d_x\}$ is a component of x, and $\alpha \in \mathbb{R}$ is a threshold. For each pair L and E, we consider the decision tree G obtained by replacing L with E in G_{cur}. The set $\text{next}(G_{\text{cur}})$ contains all G constructed in this way.

Next, the loss function $\text{cost}(G)$ is given by Eq (5.13). In each iteration, our algorithm optimizes $\text{cost}(G)$ over $G \in \text{next}(G_{\text{cur}})$, and updates $G_{\text{cur}} \leftarrow G$. To solve this optimization problem, we enumerate the possible choices $\sim$ and i and use numerical optimization to compute α (since α is a continuous parameter). An example of a single iteration of our algorithm is shown in Figure 5-5. In particular, letting G be the tree on the left and G' be the tree on the right, the left-most leaf node of G is expanded to get G'.

Our algorithm is summarized in Algorithm 1. Overall, our algorithm searches over $N \cdot (N-1) \cdot d_x$ different discrete structures, where N is the number of nodes in the

decision tree and d_x is the length of the input vector x.

5.5.7 Discussion

The adaptive teaching algorithm uses a special teacher that is similar to a student but easily optimizable. This special teacher makes both the student's learning easier (even with internal memory) and allows the teacher to adapt to the student's capabilities. One of the best parts of this algorithm, in my opinion, is the ability to synthesize each part of the state machine (i.e. the modes and the switching conditions) independently and in parallel. This compositional learning is much needed to scale the algorithm for larger state machines.

There is still much left to be done, here. There is scope for further optimizations in the teacher's and the student's algorithm. For example, currently the teacher needs to solve one optimization problem for each initial state. An alternative approach would be have a NN teacher that spits out the traces for all initial states. Another interesting aspect, here, that the student learns the modes by essentially clustering the parameters of the simple functions in the teacher's traces (Equation 5.14). An interesting question for the future is how to do this for NN based mode functions? Simple parameter clustering may not work because two neural networks with very different parameters can still have similar behaviors.

Chapter 6

Case Study — State Machine Policies for Reinforcement Learning Control Tasks

This chapter describes a case study of neurosymbolic models in reinforcement learning and evaluates their interpretability, robustness, and verifiability. This chapter is based on the results from [46] with an additional verification experiment.

Repetitive control tasks are pervasive—for example, many motor tasks such as walking, running, jumping, swimming, etc., all rely on a relatively simple behavior being repeated a certain number of times to solve the given task. However, existing deep reinforcement learning (RL) approaches have difficulty solving these tasks such that they can generalize to novel environments [72, 79]. More specifically, for a task that requires performing a repeating behavior—we would like to be able to learn a policy that generalizes to instances requiring an arbitrary number of repetitions. We refer to this property as *inductive generalization*. Moreover, we want these policies to be interpretable, modifiable and verified to show that the policy can solve the task for all possible initial state distributions.

For these purposes, state machine policies are well suited in this domain; they are sufficiently expressive to capture tasks of interest—e.g., they can perform repeating tasks by cycling through some subset of modes during execution. Additionally, state

machine policies are strongly biased towards policies that inductively generalize, that deep RL policies lack. In other words, this policy class is both *realizable* (i.e., it contains a "right" policy that solves the problem for all environments) and *identifiable* (i.e., we can learn the right policy from limited data).

We implemented our algorithm and evaluated it on reinforcement learning problems focused on tasks requiring inductive generalization. We showed that traditional deep RL approaches perform well on the original task but fail to generalize inductively, whereas our state machine policies successfully generalize beyond the training distribution.

We emphasize that we do not focus on problems that require large state machines, which is a qualitatively different problem from ours and would require different algorithms to solve. We believe that state machines are most useful when only a few modes are required. In particular, we are interested in problems where a relatively simple behavior must be repeated a certain number of times to solve the given task. The key premise behind our approach, as shown by our evaluation, is that, in these cases, compact state machines can represent policies that both have good performance and are generalizable. In fact, our algorithm solved all of our benchmarks using state machine policies with at most 4 modes. When many modes are needed, the number of possible transition structures grows exponentially, making it difficult to learn the "right" structure without exponential training data.

6.1 Tasks

We evaluated our approach on 8 control problems, each with different training and test distributions. Figure 6-1 shows the statistics regarding the benchmarks, such as the number of action variables and state variables and the set of initial states used for training and testing. Figure 6-1 also shows the different aspects of the grammar used to describe the space of possible state machine policies. We learned policies for these benchmarks using 2 to 4 distinct modes in the state machine with either a constant or a proportional grammar for the simple functions in the modes. We used a

Bench	#A	#S	X_0^{train}	X_0^{test}	# modes	M_G	C_G
Car	2	5	d ∼ [12,13.5]m	d ∼ [11,12]m	3	Constant	Boolean tree (depth 1)
Quad	1	8	x dist = 40m	x dist = 80m	2	Proportional	Boolean tree (depth 1)
QuadPO	1	4	x dist = 60m	x dist = 120m	2	Proportional	Boolean tree (depth 1)
Pendulum	1	2	mass ∼ [1,1.5]kg	mass ∼ [1.5,5]kg	2	Constant	Boolean tree (depth 2)
Cartpole	1	4	time = 5s, len = 0.5	time = 300s, len = 1.0	2	Constant	Boolean tree (depth 2)
Acrobot	1	4	masses = [0.2,0.5]	masses = [0.5,2]	2	Constant	Boolean tree (depth 2)
Mountain car	1	2	power = [5,15]e-4	power = [3,5]e-4	2	Constant	Boolean tree (depth 1)
Swimmer	3	10	len = 1 unit	len = 0.75 unit	4	Proportional	Boolean tree (depth 2)

Figure 6-1: Summary of our benchmarks. #A is the action dimension, #S is the state dimension, X_0^{train} is the set of initial states used for training, X_0^{test} is the set of initial states used to test inductive generalization, # modes is the number of modes in the state machine policy, and M_G and C_G are the grammars for the simple functions in the modes and the switching conditions, respectively. Depth of C_G indicates the number of levels in the Boolean tree.

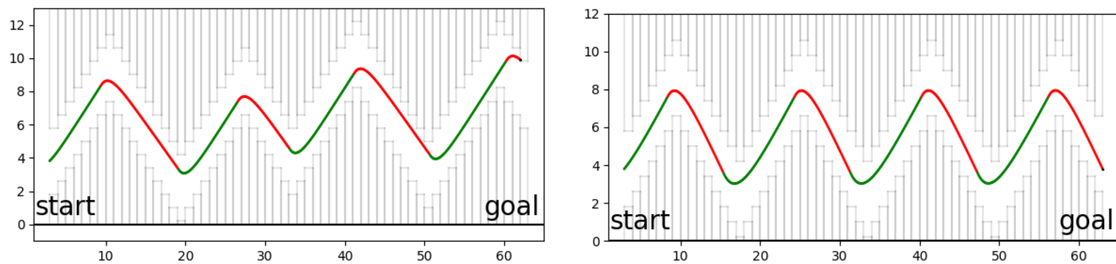


Figure 6-2: Trajectories for the Quad (left) and QuadPO (right) benchmarks using our state machine policy.

Boolean tree grammar of depth 1 or 2 for all the switching conditions. Below are the descriptions of the different benchmarks (in addition to the car task from Chapter 1):

- **Quad, QuadPO:** These benchmarks aim to maneuver a 2D quadcopter through an obstacle course by controlling its vertical acceleration (Figure 6-12). The action variable is the acceleration of the quadcopter in the vertical direction. The state for Quad includes the position x, y, the velocities v_x, v_y, and the four sensors ox_l, ox_u, oy_l, oy_u to describe the obstacle course in the immediate neighborhood. The QuadPO benchmark has the same action space as the Quad benchmark but can only observe x, y, v_x, and v_y. The functions in the different modes used for these benchmarks choose the acceleration to be proportional to v_y. To test for generalization, we vary the obstacle course length. The synthesized state machine policies for these benchmarks are shown in Figure 6-17 and Figure 6-18.

- **Pendulum:** The goal for the Pendulum benchmark is to control the force (continuous) at the actuated link to invert the link. The state includes the angle θ and the angular velocity ω of the link. We vary the pendulum mass across the train and test distributions. Figure 6-19 shows the synthesized state machine policy for the pendulum benchmark.

- **Cartpole:** The Cartpole benchmark consists of a pole attached to a cart. The goal is to keep the pole upright by applying a continuous force to move the cart to the right or the left. The state includes the position x, the velocity v of the cart, the angle θ, and the angular velocity ω of the pole. We vary the time horizon and the pole length to test for generalization. The synthesized solution is shown in Figure 6-20.

- **Acrobot:** The Acrobot benchmark is similar to the Pendulum benchmark but with two links; only the top link can be actuated, and the goal is to drive the bottom link above a certain height. The observations are the angles θ_1, θ_2 and the angular velocities ω_1, ω_2 of the two links. For this benchmark, we vary the mass of the links between the training and the test distributions. The synthesized solution is shown in Figure 6-21.

- **Mountain Car:** For the Mountain car benchmark, the goal is to drive a low powered car to the top of a hill. An agent has to go back and forth to gain enough momentum to cross the hill. The agent controls the force (continuous) to move the car to the right or left and observes the position x and the velocity v at every timestep. We vary the power of the car between the training and the test distributions. The synthesized solution is shown in Figure 6-22.

- **Swimmer:** The Swimmer benchmark is based on Mujoco's swimmer. To make this benchmark more challenging, we use 4 segments instead of 3. There are three actions that control the torques at the joints, and the goal is to make the swimmer move forward through a viscous liquid. The agent can observe the swimmer's global angle θ, the joint angles $(\theta_1, \theta_2, \theta_3)$, the swimmer's global

angular velocity ω, the angular velocities of the joints $(\omega_1, \omega_2, \omega_3)$, and the velocity of the center of mass (v_x, v_y). We vary the length of the segments between the training and the test distributions. The actions are chosen to be proportional to their corresponding angles. The synthesized state machine policy is shown in Figure 6-23.

6.2 Baselines

We compared against three baselines:

- RL: A proximal policy optimization (PPO) algorithm to train a feed-forward neural network policy
- RL-LSTM: A PPO algorithm with an LSTM model,
- Direct-Opt: learning a state machine policy directly via numerical optimization.

Hyper-parameters are chosen to maximize performance on the training distribution.

6.2.1 RL Baselines

We used the PPO2 implementation from OpenAI Baselines baselines with the standard MLP and LSTM networks for our RL baselines using 10^7 timesteps for training. Each algorithm is trained five times; we chose the one that performs best on the training distribution.

Environment featurization. We used the same action spaces, state spaces, and the set of initial states that we used for our approach. One exception is the Car benchmark, for which we appended the state vector with the state from the previous timestep. This modification was essential for the RL baseline to achieve a good performance on the training dataset.

Designing reward functions. For the classic control problems such as cartpole, pendulum, acrobot, mountain car and swimmer, we used the standard reward functions as specified by their OpenAI environments. For Quad and QuadPO benchmarks, since the goal is to avoid collisions for as long as possible, we used a reward of 1 for every timestep that the agent is alive and the agent is terminated as soon as it collides with any of the obstacles. Designing the reward function for the Car benchmark was tricky, because this benchmark has both a goal and a safety specification, and finding a right balance between them is crucial for learning. We tried various forms of rewards functions and finally, found that the following version achieves better performance on the training distribution (on the metric that measures the fraction of roll-outs that satisfy both the goal and the safety property):

$$r(x,a) = -\phi_G(x)^+ + \begin{cases} -L & \text{if } \phi_S(x) > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\phi_G(x)$ measures how close the car is to achieving its goal and $\phi_S(x) \leq 0$ represents the safe region. The above reward adds the numerical error for not satisfying the goal with a constant negative error ($-L$) if the safety specification is violated at any time step. We tried different values for $L \in \{0.1, 1, 2, 10, 20\}$ and found that $L = 10$ achieved the best performance on the training distribution.

Hyper-parameters search. We performed a search over the various hyper-parameters in the PPO2 algorithm. We ran 10 instances of the PPO2 algorithm with parameters uniformly sampled from the space given below and chose the one that performs well on the training distribution. This sampling is not exhaustive, but our results in Figure 6-3 show that we did find parameters that achieved good training performances for most of our benchmarks.

- The number of training minibatches per update,
  nminibatches $= \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048\}$.
  For the lstm network, we set this hyper-parameter to 1.

- The policy entropy coefficient in the optimization objective, $\text{ent_coef} = \{0.0, 0.01, 0.05, 0.1\}$.

- The number of training epochs per update, noptepochs $\in \{3, ..., 36\}$.

- The clipping range, cliprange $= \{0.1, 0.2, 0.3\}$.

- The learning rate, lr $\in [5 \times 10^{-6}, 0.003]$.

Note that we used model-free algorithms for the comparison to RL approaches, whereas, in our algorithm, the teacher used model-based optimization. We did not compare against model-based RL approaches because (a) even model-free RL approaches achieved almost perfect performance on the training distribution (see Figure 6-3) and (b) our main goal is to compare the performance of our policies and the neural network policies on the test distribution for generalization. Moreover, in case the model of the system is unknown, we can use known algorithms to infer the dynamics from data [1] and then use this learned dynamics in our algorithm.

6.2.2 Direct-Opt Baseline

For this baseline, we convert the problem of synthesizing a state machine policy into a numerical optimization problem. To do this, we first encode the discreteness in the grammar for switching conditions into a continuous one-hot representation. For example, the set of expressions $x[i] \leq \alpha_0$ or $x[i] \geq \alpha_0$ are encoded as $\alpha_s(\alpha_1 x[1] + \alpha_2 x[2] + \cdots \alpha_n x[n]) \leq \alpha_0$ with constraints $-1 \leq \alpha_s \leq 1$, $\forall i \in \{1, ..., n\}$. $0 \leq \alpha_i \leq 1$ and $\sum_{i=1}^{n} \alpha_i = 1$. The choices between the leaf expressions, conjunctions, and disjunctions are also encoded in a one-hot fashion. We tried another encoding without the extra constraints on α—i.e., the switching conditions are linear functions of the observations. We would expect the linear encoding to be less generalizable than the one-hot encoding. However, we found that it is hard to even synthesize a policy that works well on the training set with either of the encodings.

Another difficulty with direct optimization is that we need to optimize the combined reward from all the initial states at once. In contrast, the numerical optimiza-

tion performed by the teacher in our approach can optimize the reward for each initial state separately. To deal with the issue, we used a batch optimization technique that uses 10 initial states for every batch and seeds the starting point of the numerical optimization for each batch with the parameters found so far. We restart the process with a random starting point if the numerical optimization stalls. We carried out this process in parallel using 10 threads until either a solution was found or the time exceeded 2 hours.

6.3 Hyper-Parameters in Adaptive Teaching

There are three main hyper-parameters in our algorithm:

- The maximum number of segments/modes in a loop-free policy. A large number of segments makes the teacher's numerical optimization slow, while a small number of segments might not be sufficient to get a high reward.

- The maximum time that a segment can be executed for in a loop-free policy. This maximum time constraint helps the numerical optimization avoid local optima that arise from executing a particular (non-convex) simple policy for too long.

- The parameter λ in Section 5.4. This parameter strikes a balance between preferring high-reward loop-free policies versus preferring policies similar to the state machine learned so far.

The first two parameters solely affect the teacher's algorithm; thus, we chose them by randomly sampling from a set and selected the one that produced high-reward loop-free policies. We used $\lambda = 100$ for all our experiments.

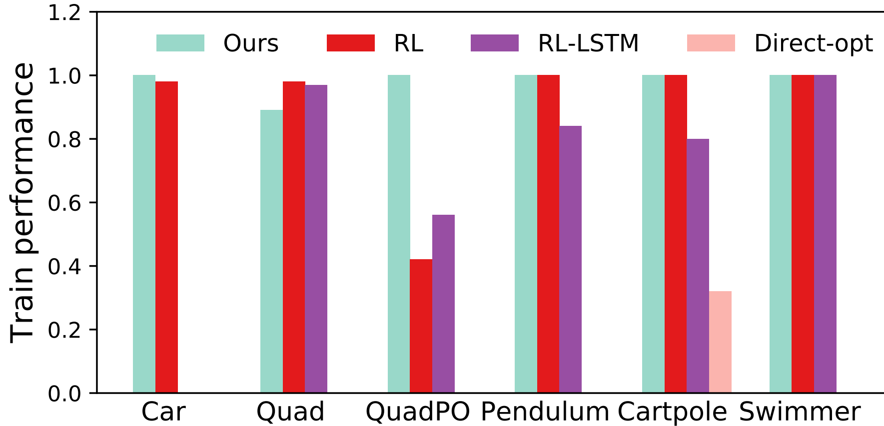


Figure 6-3: Comparison of performances on the train distribution. Our approach performs almost similar to the RL baselines, showing that our approach is expressive for these tasks. An empty bar indicates that the policy learned for that experiment failed on all runs.

6.4 Emperical Results

6.4.1 Inductive Generalization

Figure 6-3 and 6-4 show the results on the training and test distributions (respectively). We measured performance as the fraction of rollouts (out of 1000) that both satisfy the safety specification and reach the goal. Figure 6-5 shows the training and test performance for the acrobot and mountain car benchmarks.

For all benchmarks, our policy generalizes well on the test distribution. In six cases, we generalize perfectly (all runs satisfy the metric). For Quad and QuadPO, the policies resulted in collisions on some runs, but only towards the end of the obstacle course.

Comparison to RL Approaches

The RL policies mostly achieve good training performance but generalize poorly since they over-specialize to the states seen during training. The exceptions are Pendulum, Swimmer, Acrobot and Mountain Car. Even in these cases, the RL policies take longer than our state machine policies to reach the goals. Figure 6-6 qualitatively analyzes the policies learned by our approach versus RL for the Pendulum benchmark. We

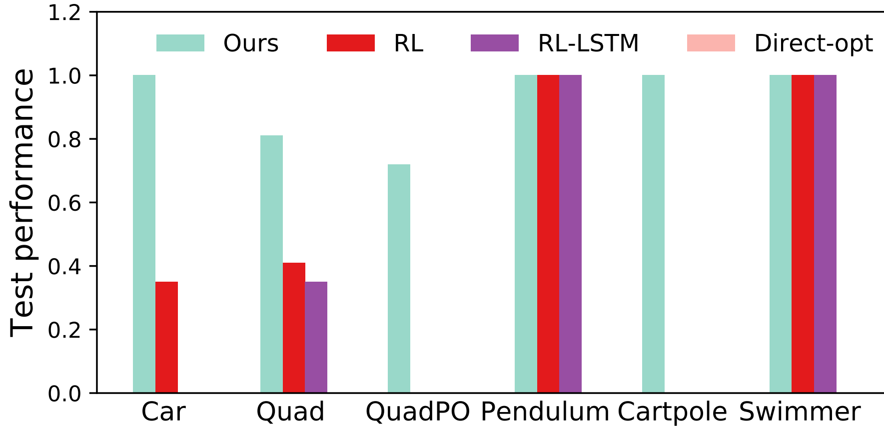


Figure 6-4: Comparison of performances on the test distribution. Our approach outperforms the baselines on all benchmarks in terms of test performance. An empty bar indicates that the policy learned for that experiment failed on all runs.

		Performance on Train dist.		Performance on Test dist.	
Bench	Algorithm	G	T_G	G	T_G
Acrobot	Ours	0.08	7.9s	0.02	31.8s
	RL	0.16	6.5s	0.0	45.2s
	Direct-opt	$\perp$	$\perp$	$\perp$	$\perp$
Mountain car	Ours	0.001	168.5s	0.008	290.1s
	RL	0.0	98.7s	0.0	214.7s
	Direct-opt	0.006	105.3s	2.18	216.0s

Figure 6-5: Experiment results for additional benchmarks. G is the average goal error (closer to 0 is better). T_G is the average number of timesteps to reach the goal (lower the better). $\perp$ indicates a timeout. We can see that both our approach and RL generalizes for these benchmarks.

can see that the RL policy performs slightly sub-optimally compared to our policy. Figure 6-7 shows the trajectories from the learned state machine policy and RL policy on Swimmer for a train environment and a test environment. While both policies generalize, the Swimmer with the state machine policy is slightly faster (it takes about 35s to cover a distance of 10 units while the RL policy takes about 45s).

For QuadPO, the RL policy does not achieve a good training performance since the states are partially observed. We may expect the LSTM policies to alleviate this issue. However, the LSTM policies often perform poorly even on the training distribution, and also generalize worse than the feed-forward neural network policies.

Figure 6-8 shows the trajectory taken by the RL policy (a), compared to our

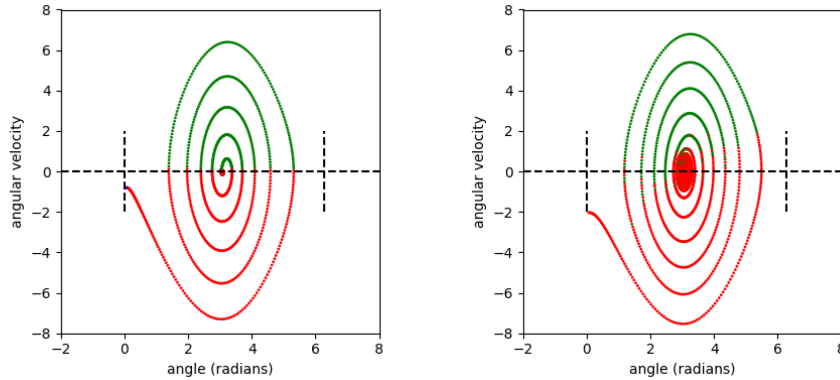


Figure 6-6: Trajectories taken by our state machine policy (left) and the RL policy (right) on Pendulum for a test environment (i.e., heavier pendulum). Green (resp., red) indicates positive (resp., negative) torque. Our policy performs optimally by using positive torque when angular velocity ≥ 0 and negative torque otherwise. In contrast, the RL policy performs sub-optimally (especially at the beginning of the trajectory).

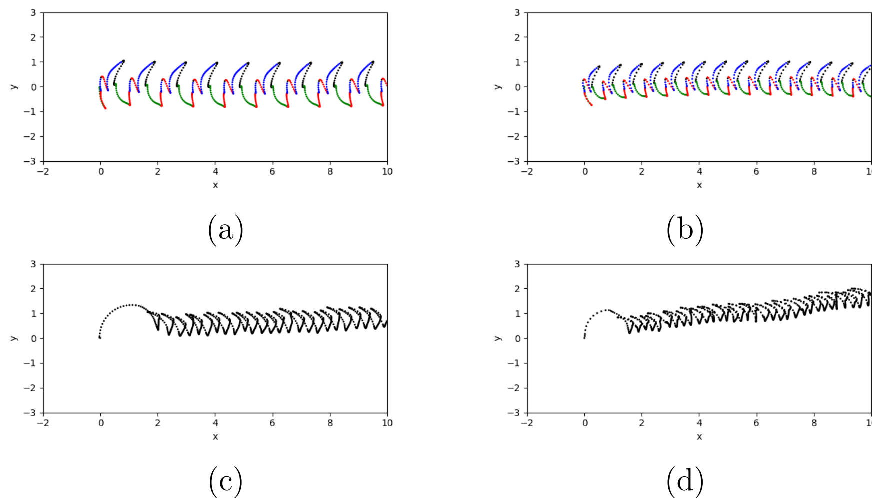
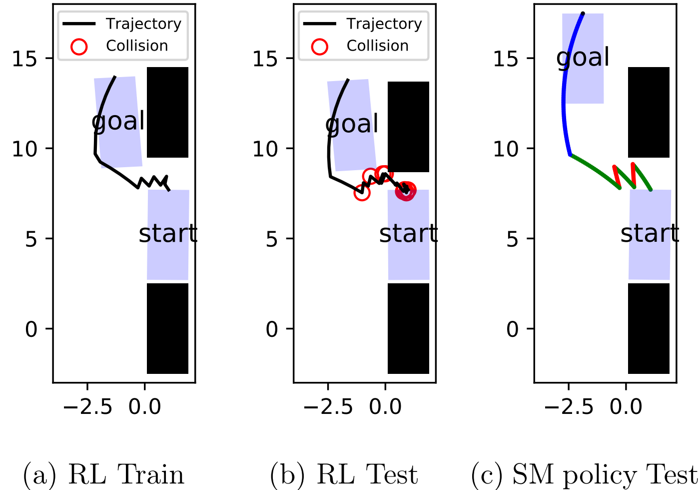


Figure 6-7: Trajectories taken by our state machine policy on Swimmer for (a) a train environment with segments of length 1, and (b) a test environment with segments of length 0.75. The colors indicate different modes. The axes are the x and y coordinates of the center of mass of the Swimmer. Trajectories taken by the RL policy on Swimmer for (c) a train environment, and (d) a test environment. While both policies generalize, the Swimmer with the state machine policy is slightly faster (it takes about 35s to cover a distance of 10 units while the RL policy takes about 45s).



(a) RL Train (b) RL Test (c) SM policy Test

Figure 6-8: The RL policy generates unstructured trajectories, and therefore does not generalize from (a) the training distribution to (b) the test distribution. In contrast, our state machine policy in (c) generates a highly structured trajectory that generalizes well.

policy (c), from a training initial state for the Car benchmark. The RL policy does not exhibit a repeating behavior, which causes it to fail on the trajectory from a test state shown in (b).

Comparison to Direct Optimization Baseline

The state machine policies learned using direct-opt baseline perform poorly even in training because of the numerous local optima arising due to the structural constraints. This result supports the need to use adaptive teaching to learn state machine policies.

Varying the Training Distribution

We, next, study how the test performance changes as we vary the training distribution on the Car benchmark. We varied X_0^{train} as $d \sim [d_{\min}, 13]$, where $d_{\min} = \{13, 12.5, 12, 11.5, 11.2, 11\}$, but fix X_0^{test} to $d \sim [11, 12]$. Figure 6-9 shows how the test performance varies with $d_{\min}$ for both our policy and the RL policy. Our policy inductively generalizes for a wide range of training distributions. In contrast, the test performance of the RL policy initially increases as the train distribution gets

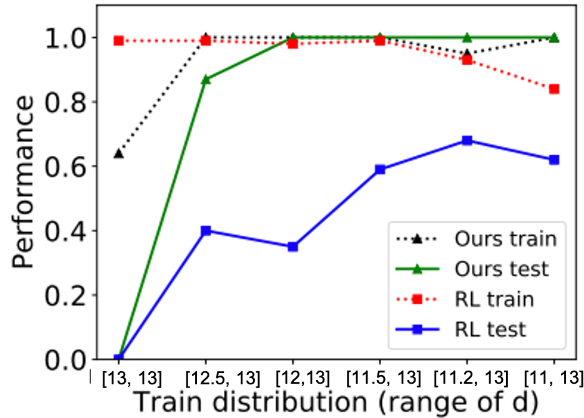
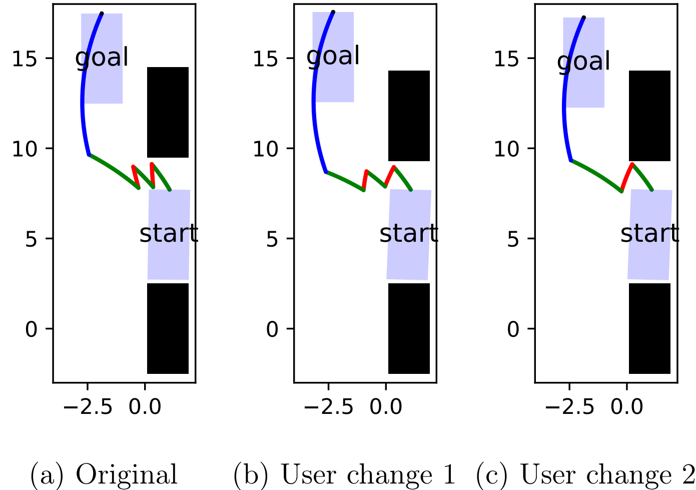


Figure 6-9: We plot the train and the test performance for different choices of training distribution for the Car benchmark .

bigger, but it eventually starts declining. The reason is that its training performance actually starts to decline. Thus, in some settings, our approach (even when trained on smaller distributions) can produce policies that outperform the neural network policies produced by RL (even when trained on the full distribution).

6.4.2 Interpretability

An added benefit of our state machine policies is interpretability. In particular, we demonstrated the interpretability of our policies by showing how a user can modify a learned state machine policy. Consider the policy from Figure 1-3 for the autonomous car. We manually made the following changes: (i) increased the steering angle in H_{m_1} to its maximum value 0.5, and (ii) decreased the gap maintained between the agent and the black cars by changing the switching condition $G_{m_1}^{m_2}$ to $d_f \leq 0.1$ and $G_{m_2}^{m_1}$ to $d_b \leq 0.1$. Figure 6-10 demonstrates these changes—it shows the trajectories obtained using the original policy (a), the first modified policy (b), and the second modified policy (c). There is no straightforward way to make these kinds of changes to a neural network policy.



(a) Original (b) User change 1 (c) User change 2

Figure 6-10: A user can modify a learned state machine policy to improve performance. In (b), the user sets the steering angle in Figure 1-3 to the maximum value 0.5, and in (c), the user sets the thresholds in the switching conditions $G_{m_1}^{m_2}, G_{m_2}^{m_1}$ to 0.1.

6.4.3 Verification

Another main advantage of neurosymbolic policies is that they are significantly easier to verify formally. Intuitively, because they effectively use discrete control flow structures, it is easier for formal methods to prune branches of the search space corresponding to unreachable program paths. A standard strategy for verifying safety is to devise a logical formula that encodes a trajectory rollout and the safety constraint at every timestep. Then, feed this logical formula to a Satisfiability Modulo Theory (SMT) [23] solver to check for safety violations.

As an example, we used this strategy to verify that the state machine policy (in Figure 1-3) for the car task in Figure1-1 is correct—i.e., it successfully exits the parking spot without colliding with the other cars. In this case, we used a safety verification tool for hybrid systems called dReach [55] to encode and solve this verification problem. dReach can handle general hybrid systems with nonlinear differential equations and complex discrete mode-changes. Thus, it is a good fit for our situation. dReach performs bounded reachability analysis; it can verify up to some pre-specified bound on the unrolling of the state machine modes. dReach is "δ complete," i.e. when "safe" is the answer, we know for sure that the system does not reach the unsafe

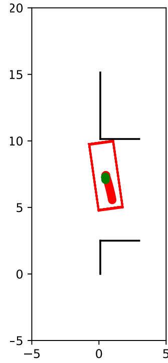


Figure 6-11: A failure case found with verification with a noise of 0.24 in the environment, where the car collides with the car in the front.

region; when "δ-unsafe" is the answer, there exists some δ-bounded perturbation in the system that would render it unsafe. Thus, we could vary δ to measure how robust a system is to perturbations.

With $\delta = 0.1$, the dReach solver proved that the policy in Figure 1-3 is indeed safe for up to 7 mode unrolling of the state machine (which covers a significant fraction of the initial state space; the rest timed out). However, with $\delta = 0.24$, the dReach solver identified a failure case where the car would collide with the car in the front (under some perturbations of the original model), as shown in Figure 6-11 This problem can be fixed by manually examining the state machine policy and modifying the switching conditions $G_{m_1}^{m_2}$ to $d_f \leq 0.5$ and $G_{m_2}^{m_1}$ to $d_b \leq 0.5$. With these changes, the policy is now correct even for $\delta = 0.24$.

6.4.4 Behavior of Policy

We empirical analyze the behavior of the learned policies. Figure 6-12 (right) compares the actions taking by our policy to those taken by the RL policy on Quad and QuadPO. Our policy produces smooth repeating actions, whereas the RL policy does not. This example further illustrates how neurosymbolic policies are both complex (evidenced by the complexity of the red curve) yet structured (evidenced by the smoothness of the red curve and its repeating pattern). In contrast, DNN policies are

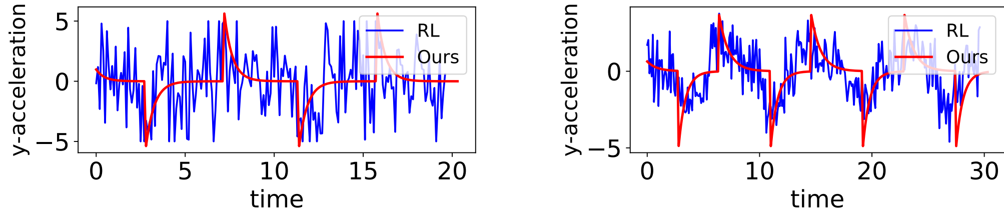


Figure 6-12: Graph of vertical acceleration over time for both our policy (red) and the neural network policy (blue) for Quad (left) and QuadPO (right).

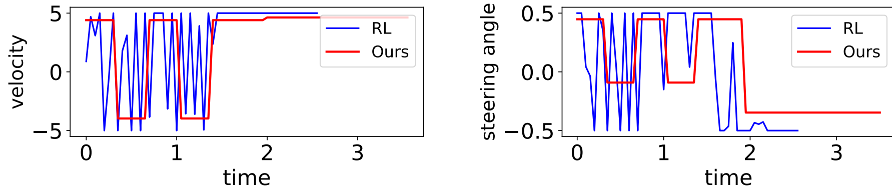


Figure 6-13: Action vs time graphs for the car benchmark for our policy (red) and the neural network policy (blue). (Left) shows the velocity of the agent, and (Right) shows the steering angle.

expressive (as evidenced by the complexity of the red curve) but lack the structure needed to generalize robustly.

Figures 6-13, 6-14, & 6-15 show the action versus time plots for the various benchmarks using the learned state machine policies and neural network policies. Even here, we can see that state machine polices produce smooth actions, whereas the RL policies do not.

6.4.5 Analysis of Running Time

Figure 6-16 shows the synthesis times for various benchmarks. It also shows the number of student-teacher iterations and the time spent separately by the teacher and the student. The teacher optimizes the loop-free policies for different initial states in parallel. The student optimizes the switching conditions between different pairs of modes in parallel. We used a parallelized implementation with 10 threads and reported the wall clock time.

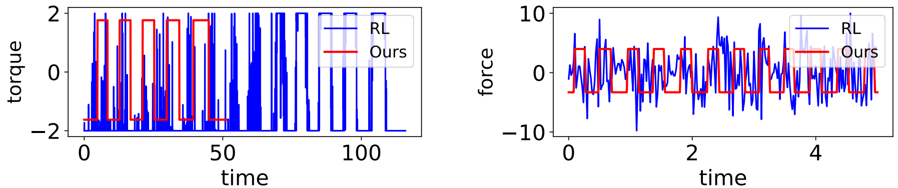


Figure 6-14: Action vs time graphs for the pendulum benchmark (left) and the cart-pole benchmark (right) for both our policy (red) and the neural network policy (blue).

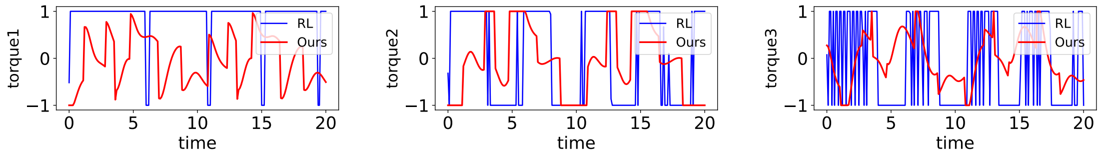


Figure 6-15: Action vs time graphs for the swimmer benchmark for the three torques at the three different joints of the swimmer. The blue line is for the neural network policy, and the red is for the state machine policy.

6.5 Discussion

In this case study, we learned state machine policies for control tasks requiring repetitive behaviors. Our learning approach is based on a framework called adaptive teaching that alternatively learns a student that imitates a teacher, who in-turn adapts to the structure of the student. We demonstrated that our policies inductively generalize better than traditional RL policies.

Future works include exploring more complex grammars for the simple functions and the switching conditions, for example, with some parts being small neural networks, while still retaining the ability to learn generalizable behaviors. These more complex grammars are necessary for complex tasks and to handle high-dimensional inputs such as images. Another direction is to extend our approach to use model-free techniques in the teacher's algorithm to make our approach more aligned with the reinforcement learning premise. Finally, I believe that the idea of learning neurosymbolic state machines and using the adaptive teaching algorithm to deal with the mixed discrete-continuous problems can be applied to other learning settings and domains.

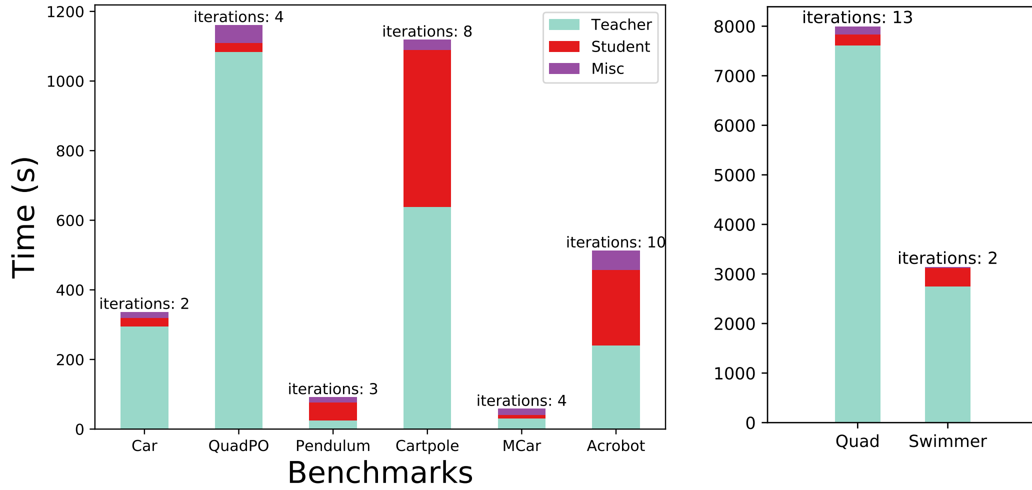


Figure 6-16: Synthesis times (in seconds, wall clock time) for learning state machines policies for the different benchmarks. The plot breaks down the total synthesis time into the time taken by the teacher, the student and other miscellaneous parts of the algorithm. Misc. mainly includes the time spent for checking convergence at every iteration. The plot also shows the number of teacher-student iterations taken for each benchmark.

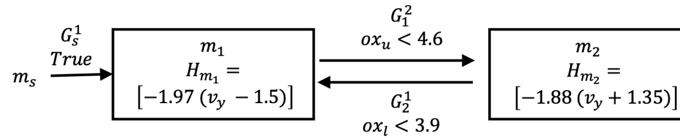


Figure 6-17: Synthesized state machine policy for the Quad benchmark.

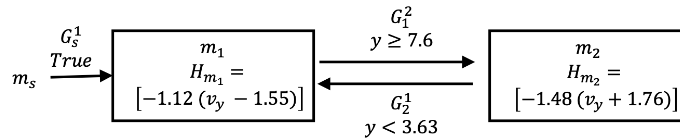


Figure 6-18: Synthesized state machine policy for the QuadPO benchmark.

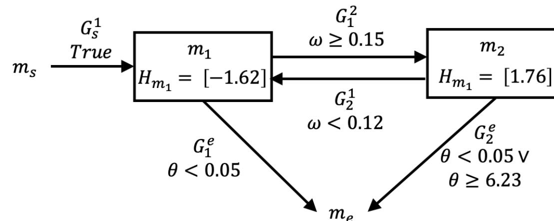


Figure 6-19: Synthesized state machine policy for Pendulum.

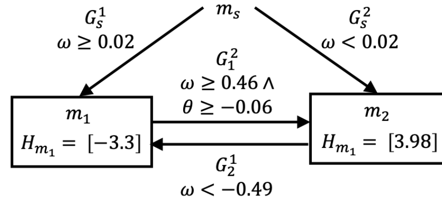


Figure 6-20: Synthesized state machine policy for Cartpole.

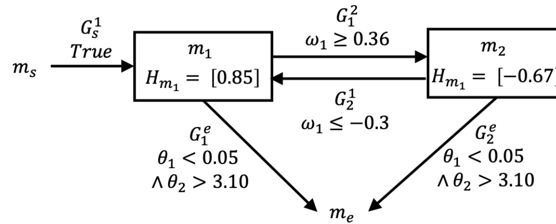


Figure 6-21: Synthesized state machine policy for Acrobot.

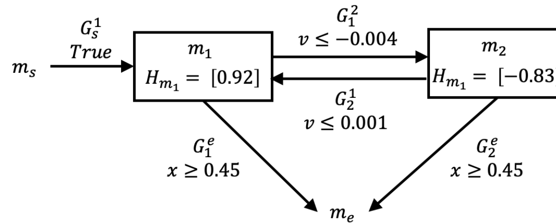


Figure 6-22: Synthesized state machine policy for Mountain car.

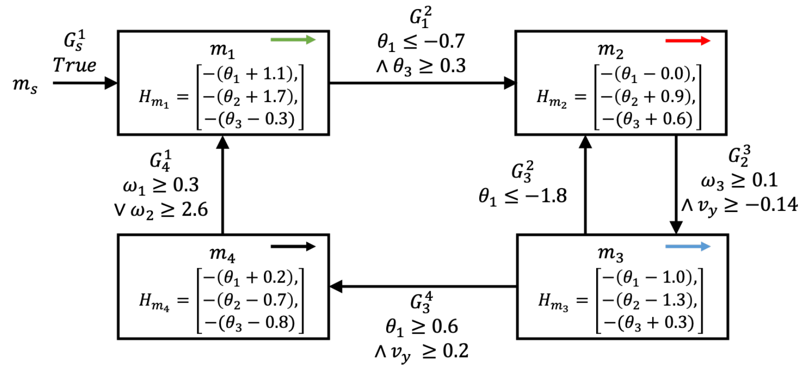


Figure 6-23: Synthesized state machine policy for Swimmer.

Chapter 7

Case Study: Neurosymbolic Transformers for Multi-Agent Communications

This chapter presents the second case study of neurosymbolic models in the context of multi-agent systems. This chapter is based on the results from [47] with an additional generalization experiment.

Many real-world robotics systems are distributed, with teams of agents needing to coordinate to share information and solve problems. Reinforcement learning has recently been demonstrated as a promising approach to automatically solve such multi-agent planning problems [94, 61, 35, 62, 36, 53].

A key challenge in (cooperative) multi-agent planning is how to coordinate with other agents, both deciding whom to communicate with and what information to share. One approach is to let agents communicate with all other agents; however, letting agents communicate arbitrarily can lead to poor generalization [52, 74]; furthermore, it cannot account for physical constraints such as limited bandwidth. A second approach is to manually impose a communication graph on the agents, typically based on distance [52, 95, 74, 87]. However, this manual structure may not reflect the optimal communication structure—for instance, one agent may prefer to communicate with another one that is farther away but in its desired path. A third

approach is to use a transformer [98] as the policy network [22], which uses attention to choose which other agents to focus on. However, since the attention is soft, each agent still communicates with every other agent.

In this case study, we study the problem of learning a communication policy that solves a multi-agent planning task while minimizing the amount of communication required. We measure the amount of communication on a given step as the maximum degree (in both directions) of the communication graph on that step; this metric captures the maximum amount of communication any single agent must perform at that step. While we focus on this metric, our approach easily extends to handling other metrics—e.g., the total number of edges in the communication graph, the maximum in-degree, and the maximum out-degree, as well as general combinations of these metrics.

A key question is how to represent the communication policy; in particular, it must be sufficiently expressive to capture communication structures that both achieve high reward and has low communication degree, while simultaneously being easy to train. Neural network policies can likely capture good communication structures, but they are hard to train since the maximum degree of the communication graph is a discrete objective that cannot be optimized using gradient descent. An alternative is to use a structured model such as a decision tree [14] or rule list [103] and train using combinatorial optimization. However, these models perform poorly since choosing whom to communicate with requires reasoning over sets of other agents—e.g., to avoid collisions, an agent must communicate with its nearest neighbor in its direction of travel.

Therefore, we need domain-specific programs to represent communication policies. In contrast to rule lists, our programmatic polices include components such as filter and map that operate over sets of inputs. Furthermore, programmatic policies are discrete in nature, making them amenable to combinatorial optimization; in particular, we can compute a programmatic policy that minimizes the communication graph degree using a stochastic synthesis algorithm [83] based on MCMC sampling [66, 43].

A key aspect of our programs is that they can include a random choice operator.

Intuitively, random choice is a key ingredient needed to minimize the communication graph degree without global coordination. For example, suppose there are two groups of agents, and each agent in group A needs to communicate with an agent in group B, but the specific one does not matter. Using a deterministic communication policy, since the same policy is shared among all agents, each agent in group A might choose to communicate with the same agent j in group B (e.g., if agents in the same group have similar states). Then, agent j will have a very high degree in the communication graph, which is undesirable. In contrast, having each agent in group A communicate with a uniformly random agent in group B provides a near-optimal solution to this problem, without requiring the agents to explicitly coordinate their decisions.

While we can minimize the communication graph degree using stochastic search, we still need to choose actions based on the communicated information. Hence, neurosymbolic transformers are good candidates for multi-agent planning problems.

We evaluated our approach on several multi-agent planning tasks that require agents to coordinate to achieve their goals. We demonstrated that our algorithm learns communication policies that achieve task performance similar to the original transformer policy (i.e., where each agent communicates with every other agent), while significantly reducing the amount of communication. Our results showed that our algorithm is a promising approach for training policies for multi-agent systems that additionally optimize combinatorial properties of the communication graph [1]

7.1 Multi-Agent RL Related work

There has been a great deal of recent interest in using reinforcement learning to automatically infer good communication structures for solving multi-agent planning problems [52, 95, 74, 22, 87]. Much of this work focuses on inferring what to communicate rather than whom to communicate with; they handcraft the communication structure to be a graph (typically based on distance) [52, 95, 74], and then use a graph

[1] The code and a video illustrating the different tasks are available at `https://github.com/jinala/multi-agent-neurosym-transformers`.

neural network [81, 54] as the policy network. There has been some prior work using transformer networks to infer the communication graph [22]; however, they rely on soft attention, so the communication graph remains fully connected. Prior work [89] frames the multi-agent communication problem as an MDP problem where the decisions of when to communicate are part of the action space. However, in our case, we want to learn who to communicate with in addition to when to communicate. This results in a large discrete action space, and we found that RL algorithms perform poorly in this space. Our proposed approach addresses this challenge by using the transformer as a teacher.

7.2 Multi-Agent Problem Formulation

We formulate the multi-agent planning problem as a decentralized partially observable Markov decision process (POMDP). We consider N agents $i \in [N] = \{1, ..., N\}$ with states $s^i \in \mathcal{S} \subseteq \mathbb{R}^{d_S}$, actions $a^i \in \mathcal{A} \subseteq \mathbb{R}^{d_A}$, and observations $o^{i,j} \in \mathcal{O} \subseteq \mathbb{R}^{d_O}$ for every pair of agents ($j \in [N]$). Following prior work [62, 36, 22], we operate under the premise of centralized training and decentralized execution. Hence, during training the POMDP has global states $\mathcal{S}^N$, global actions $\mathcal{A}^N$, global observations $\mathcal{O}^{N\times N}$, transition function $F : \mathcal{S}^N \times \mathcal{A}^N \to \mathcal{S}^N$, observation function $Z : \mathcal{S}^N \to \mathcal{O}^{N\times N}$, initial state distribution $s_0 \sim \mathcal{P}_0$, and reward function $r : \mathcal{S}^N \times \mathcal{A}^N \to \mathbb{R}$.

The agents all use the same policy $\pi = (\pi^C, \pi^M, \pi^A)$ divided into a *communication policy* π^C (choose other agents from whom to request information), a *message policy* π^M (choose what messages to send to other agents), and an *action policy* π^A (choose what action to take). Below, we describe how each agent $i \in [N]$ chooses its action a^i at any time step.

Step 1 (Choose communication). The communication policy $\pi^C : \mathcal{S} \times \mathcal{O}^N \to \mathcal{C}^K$ inputs the state s^i of current agent i and its observations $o^i = (o^{i,1}, ..., o^{i,N})$, and outputs K other agents $c^i = \pi^C(s^i, o^i) \in \mathcal{C}^K = [N]^K$ from whom to request information. The *communication graph* $c = (c^1, ..., c^N) \in \mathcal{C}^{N\times K}$ is the directed graph $G = (V, E)$ with nodes $V = [N]$ and edges $E = \{j \to i \mid (i, j) \in [N]^2 \wedge j \in \pi^C(s^i, o^i)\}$.

Step 2 (Choose and send/receive messages). For every other agent $j \in [N]$, the message policy $\pi^M : \mathcal{S} \times \mathcal{O} \to \mathcal{M}$ inputs s^i and $o^{i,j}$ and outputs a message $m^{i \to j} = \pi^M(s^i, o^{i,j})$ to be sent to j if requested. Then, agent i receives messages $m^i = \{m^{j \to i} \mid j \in c^i\} \in \mathcal{M}^K$.

Step 3 (Choose action). The action policy $\pi^A : \mathcal{S} \times \mathcal{O}^N \times \mathcal{M}^K \to \mathcal{A}$ inputs s^i, o^i, and m^i, and outputs action $a^i = \pi^A(s^i, o^i, m^i)$ to take.

Here, each agent computes its action based on just its state, its observations of other agents, and communications received from the other agents; thus, the policy can be executed in a decentralized way.

Sampling a trajectory/rollout. Given initial state $s_0 \sim \mathcal{P}_0$ and time horizon T, π generates the trajectory $(s_0, s_1, ..., s_T)$, where $o_t = Z(s_t)$ and $s_{t+1} = F(s_t, a_t)$, and where for all $i \in [N]$, we have $c_t^i = \pi^C(s_t^i, o_t^i)$, $m_t^i = \{\pi^M(s_t^j, o_t^{j,i}) \mid j \in c_t^i\}$, and $a_t^i = \pi^A(s_t^i, o_t^i, m_t^i)$.

Objective. Then, our goal is to train a policy π that maximizes the objective

$$J(\pi) = J^R(\pi) + \lambda J^C(\pi) = \mathbb{E}_{s_0 \sim \mathcal{P}_0}\left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t)\right] - \lambda \mathbb{E}_{s_0 \sim \mathcal{P}_0}\left[\sum_{t=0}^{T} \max_{i \in [N]} \deg(i; c_t)\right] \tag{7.1}$$

where $\lambda \in \mathbb{R}_{>0}$ is a hyperparameter, the reward objective J^R is the time-discounted expected cumulative reward over time horizon T with discount factor $\gamma \in (0, 1)$, and the communication objective J^C is to minimize the degree of the communication graph, where c_t is the communication graph on step t, and $\deg(i; c_t)$ is the sum of the incoming and outgoing edges for node i in c_t.

Assumptions on the observations of other agents. We assume that $o^{i,j}$ is available through visual observation (e.g., camera or LIDAR), and therefore does not require extra communication. In all experiments, we use $o^{i,j} = x^j - x^i + \epsilon^{i,j}$—i.e., the position x^j of agent j relative to the position x^i of agent i, plus i.i.d. Gaussian noise $\epsilon^{i,j}$. This information can often be obtained from visual observations (e.g., using an

object detector); $\epsilon^{i,j}$ represents noise in the visual localization process.

The observation $o^{i,j}$ is necessary since it forms the basis for agent i to decide whether to communicate with agent j; if it is unavailable, then i has no way to distinguish the other agents. If $o^{i,j}$ is unavailable for a subset of agents j (e.g., they are outside of sensor range), we could use a mask to indicate that the data is missing. We could also replace it with alternative information such as the most recent message from j or the most recent observation of j.

We emphasize that $o^{i,j}$ does not contain important internal information available to the other agents—e.g., their chosen goals and their planned actions/trajectories. This additional information is critical for the agents to coordinate their actions and the agents must learn to communicate such information.

7.3 Neurosymbolic Transformers for Multi-Agent problem

The above policy description for the multi-agent systems can easily be encoded using a transformer architecture (and hence, a neurosymbolic transformer). The communication policy is the attention network which is encoded by the key and the query networks in a transformer (the attention program in a neurosymbolic transformer). The message computation step is the same as the value network in the transformer, and similarly, the action computation step is the same as the output network in the transformer. We use the following grammar for the rules in the attention programs:

$$R \ ::= \ \mathsf{argmax}((F, \mathsf{filter}(B, \ell))) \mid \mathit{choose}(\mathsf{filter}(B, \ell)).$$

Intuitively, the first kind of rule is a deterministic aggregation rule, which uses F to score every agent after filtering and then chooses the one with the best score, and the second kind of rule is a nondeterministic choice rule which randomly chooses one of the other agents after filtering.

7.4 Tasks

Formation task. We consider multi-agent formation flying tasks in 2D space [52]. Each agent has a starting position and an assigned goal position. The task is to learn a decentralized policy for the agents to reach the goals while avoiding collisions. The agents are arranged into a small number of groups (between 1 and 4): starting positions for agents within a group are close together, as are goal positions. Each agent's state s^i contains its current position x^i and goal position g^i. The observations $o^{i,j} = x^j - x^i + \epsilon^{i,j}$ are the relative positions of the other agents, corrupted by i.i.d. Gaussian noise $\epsilon^{i,j} \sim \mathcal{N}(0, \sigma^2)$. The actions a^i are agent velocities, subject to $\|a^i\|_2 \leq v_{\max}$. The reward at each step is $r(s,a) = r^g(s,a) - r^c(s,a)$, where the goal reward $r^g(s,a) = -\sum_{i \in [N]} \|x^i - g^i\|_2$ is the negative sum of distances of all agents to their goals, and the collision penalty $r^c(s,a) = \sum_{i,j \in [N], i \neq j} \max\{p_c(2 - \|x^i - x^j\|_2/d_c), 0\}$ is the hinge loss between each pair of agents, where p_c is the collision penalty weight and d_c is the collision distance.

We consider two instances of formation tasks:

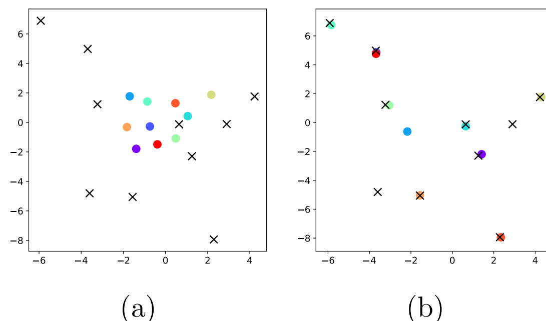
First, `random-cross`, which contains up to 4 groups; each possible group occurs independently with probability 0.33. The starting positions in each group (if present) are sampled uniformly randomly inside 4 boxes with center b equal to $(-\ell, 0)$, $(0, -\ell)$, $(\ell, 0)$, and $(0, \ell)$, respectively, and the goal positions of each group are sampled randomly from boxes with centers at $-b$. The challenge is that agents in one group must communicate with agents in other groups to adaptively choose the most efficient path to their goals.

Second, `random-grid` (Figure 1-4a) contains 3 groups with starting positions sampled in boxes centered at $(-\ell, 0)$, $(0, 0)$, and $(\ell, 0)$, respectively, and the goal positions are sampled in boxes centered at randomly chosen positions $(b_x, b_y) \in \{-\ell, 0, \ell\}^2$ (i.e., on a 3×3 grid), with the constraint that the starting box and goal box of a group are adjacent and the boxes are all distinct. The challenge is that each agent must learn whom to communicate with depending on its goal.

Unlabeled goals task. This task is a cooperative navigation task with unlabeled

goals [62] that has N agents along with N goals at positions $g_1, ..., g_N$ (see Figure 7-1). The task is to drive the agents to cover as many goals as possible. We note that this task is not just a navigation task. Since the agents are not pre-assigned to goals, there is a combinatorial aspect where they must communicate to assign themselves to different goals. The agent state s^i is its own position x^i and the positions of the goals (ordered by distance at the initial time step). The observations $o^{i,j}$ are the relative positions to the other agents, corrupted by Gaussian noise. The actions $a^i = (p^i_1, \cdots, p^i_l, \cdots, p^i_N)$ are the weights (normalized to 1) over the goals; the agent moves in the direction of the weighted sum of goals—i.e., its velocity is $a^i = \sum_{k\in[N]} p^i_k(g_k - x^i)$. The reward is $r(s, a) = \sum_{k\in[N]} \max_{i\in[N]} p^i_k - N$—i.e., the sum over goals of the maximum weight that any agent assigns to that goal minus N.

The code and a short video illustrating the different tasks used in the paper can be found at
`https://github.com/jinala/multi-agent-neurosym-transformers`.



(a) (b)

Figure 7-1: Unlabeled goals task: (a) Initial positions of the agents and the locations of the goals to cover (b) Final configuration of the agents where 8 out of the 10 goals are covered.

7.5 Baselines

We consider the following baselines.

- Fixed communication (`dist`): A transformer, but where other agents are masked based on distance, so each agent can only attend to its k nearest neighbors.

We found this model outperforms GCNs with the same communication structure [52], since its attention parameters enable each agent to re-weight the messages it receives.

- Transformer (`tf-full`): The oracle transformer policy from Section 3.2.1; here, each agent communicates with all other agents.

- Transformer + hard attention (`hard-attn`): The transformer policy, but where the communication degree is reduced by constraining each agent to only receive messages from k other agents with the largest attention scores. Note that this approach only minimizes the maximum in-degree, not necessarily the maximum out-degree; minimizing both would require a centralized algorithm.

- Transformer + program (`prog`): An ablation of our approach that does not retrain the transformer after synthesizing the programmatic communication policy

- Transformer + retrained program (`prog-retrain`): Our full approach.

- Transformer + deterministic programs (`det-prog` and `det-prog-retrained`): ablations of our approach that learns only deterministic rules—i.e., rules with `random` are excluded from the search space, before and after the retraining step.

- Transformer + decision trees (`dt` and `dt-retrained`): a learned communication policy in the form of a decision tree. To train the decision tree, we constructed a supervised dataset by (i) collecting the soft-attentions from the transformer model, and (ii) solving the global hard-attention problem at each timestep to ensure that the maximum degree (both in-degree and out-degree) is at most k, where k is chosen as described below (i.e., to match the number of rules in our programmatic communication structure). Then, we trained the decision tree using supervised data on this dataset.

The tasks `random-cross` and `random-grid` perform 1 round of communications per time step (i.e., 1 attention layer) for all the baselines. The `unlabeled-goals` task uses 2 rounds of communications (i.e., 2 attention layers). For all approaches, we trained the oracle transformer model with 10k rollouts. For synthesizing the programmatic policy, we built a dataset using 300 rollouts and ran MCMC for 10000 steps. We retrained the transformer with 1000 rollouts. We constrained the maximum in-degree to be a constant d_0 across all approaches (except `tf-full`, where each agent communicates with every other agent); for `dist` and `hard-attn`, we did so by setting the communication neighbors to be $k = d_0$, and for `prog` and `prog-retrain`, we chose the number of rules to be $K = d_0$. This choice ensures fair comparison across approaches.

7.6 Hyper-parameters

There are four main hyper-parameters in our learning algorithm.

- $\tilde{\lambda}$ in Section 5.2.1: This parameter strikes a balance between minimizing the difference in the actions (with and without programmatic attention) and minimizing the maximum communication degree. We sampled $\tilde{\lambda}$ from $\{0.3, 0.5, 0.7, 1.0\}$.
- The number of rules in the program $\in \{2, 3, 4, 5\}$.
- The depth of the Boolean conditions in the filter expressions $= 2$.
- The feature map ϕ used in the filter predicates and the map functions. We have 2 versions: 1) for every vector (x, y) in the state s and the observations o, we also encode the norm $\sqrt{x^2 + y^2}$ and the angle $\tan^{-1}(y/x)$ as part of the features; 2) on top of 1, we add quadratic features $(x_s x_o, x_s y_o, y_s x_o, y_s y_o)$ where (x_s, y_s) is the state and (x_o, y_o) is the observation.

We used cross validation to choose these parameters. In particular, we chose the ones that produced the lowest cumulative reward on a validation set of rollouts; if the cumulative rewards are similar, we chose the ones that reduced the communication degree.

7.7 Emperical Results

7.7.1 Performance and Expressiveness

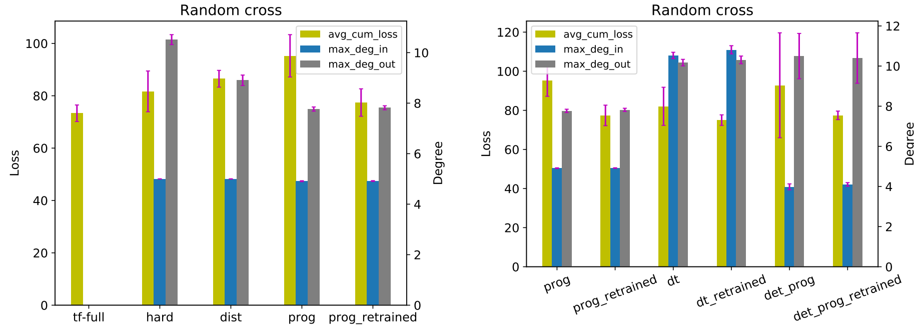
We measured performance using both the loss (i.e., negative reward) and maximum communication degree (i.e., maximum degree of the communication graph), averaged over the time horizon. Lower values are better for both the metrics. Because the in-degree of every agent is constant, the maximum degree equals the in-degree plus the maximum out-degree. Thus, we report the maximum in-degree and the maximum out-degree separately. Results are in Figure 7-2; we report mean and standard deviation over 20 random seeds.

For `random-cross` and `random-grid` tasks, our approach (`prog-retrained`) achieves loss similar to the best loss (i.e., that achieved by the full transformer), while simultaneously achieving the best communication graph degree. In general, approaches that learn communication structure using attention (`tf-full`, `hard-attn`, `prog-retrained`, and `dt-retrained`) perform better than having a fixed communication structure (i.e., `dist`). In addition, using the programmatic attention is more effective at reducing the maximum degree (in particular, the maximum out-degree) compared with thresholding the transformer attention (i.e., `hard-attn`). Finally, retraining the transformer is necessary for the neurosymbolic transformer to perform well in terms of loss.

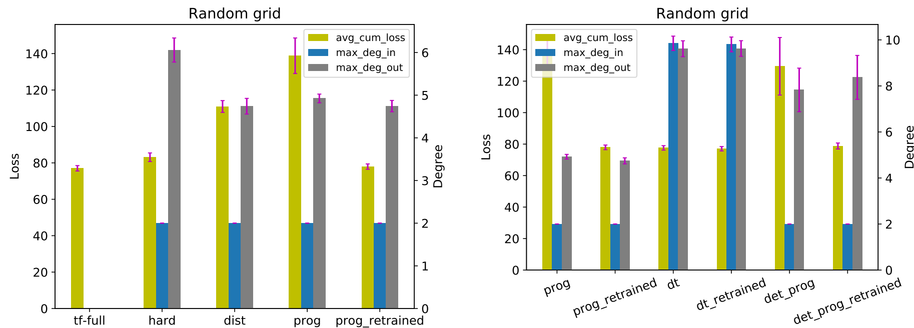
For `unlabeled-goals` task, our approach performs almost similar to `dist` baseline and slightly worse than `tf-full` baseline, but achieves a smaller communication degree. Moreover, the loss is significantly lower than the loss of 4.13 achieved when no communications are allowed.

The decision tree baselines (`dt` and `dt-retrained`) perform poorly in terms of the communication degree for all the tasks, demonstrating that domain-specific programs operating over lists are necessary to reduce the communications.

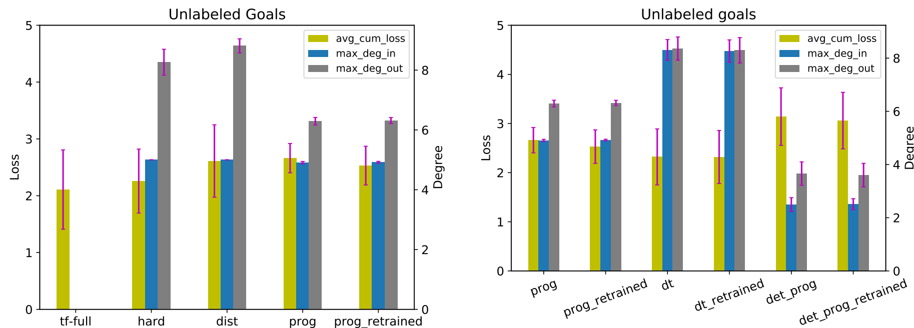
The deterministic baseline (`det-prog-retrained`) achieves a similar loss as `prog-retrained` for the `random-cross` and `random-grid` tasks; however, it has worse out-degrees of communication. For these tasks, it is difficult for a de-



(a)



(b)



(c)

Figure 7-2: Statistics of cumulative loss and communication graph degrees across baselines, for (a) random-cross, (b) random-grid, and (c) unlabeled-goals. We omit communication degrees for `tf-full`, since it requires communication between all pairs of agents.

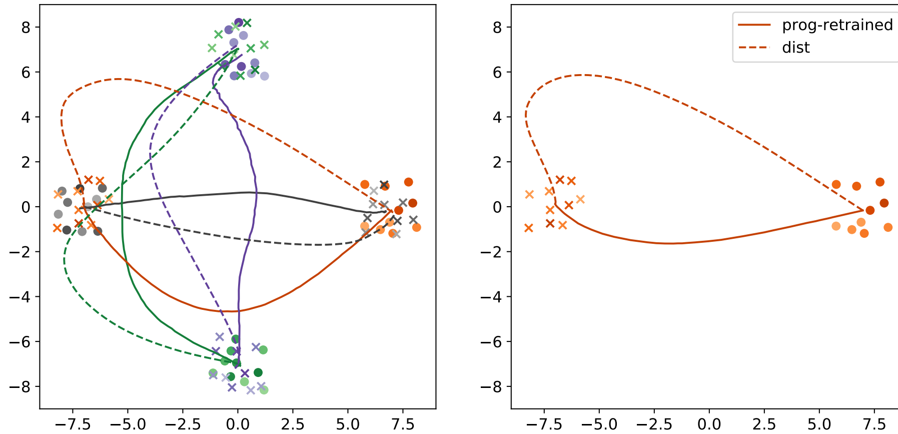


Figure 7-3: For `random-cross`, trajectories taken by each group (i.e., averaged over all agents in that group) when all four groups are present (left) and only one group is present (right), by `prog-retrained` (solid) and `dist` (dashed). Initial positions are circles and goal positions are crosses.

terministic program to distinguish the different agents in a group; thus, all agents request messages from a small set of agents. For the `unlabeled goals` task, the deterministic baseline has a lower degree of communication but has higher loss than `prog-retrained`. Again, we hypothesize that the deterministic rules are insufficient for an agent to distinguish the other agents, leading to a low in-degree (and consequently low out-degree), which is insufficient to solve the task.

Analyzing the Learned Policies

Figure 7-3 shows two examples from the `random-cross` task: all four groups are present (left), and only a single group is present (right). In the former case, the groups must traverse complex trajectories to avoid collisions, whereas in the latter case, the single group can move directly to the goal. However, with a fixed communication structure, the policy `dist` cannot decide whether to use the complex trajectory or the direct trajectory since it cannot communicate with agents in other groups to determine if it should avoid them. Thus, it always takes the complex trajectory. In contrast, our approach successfully decides between the complex and direct trajectories.

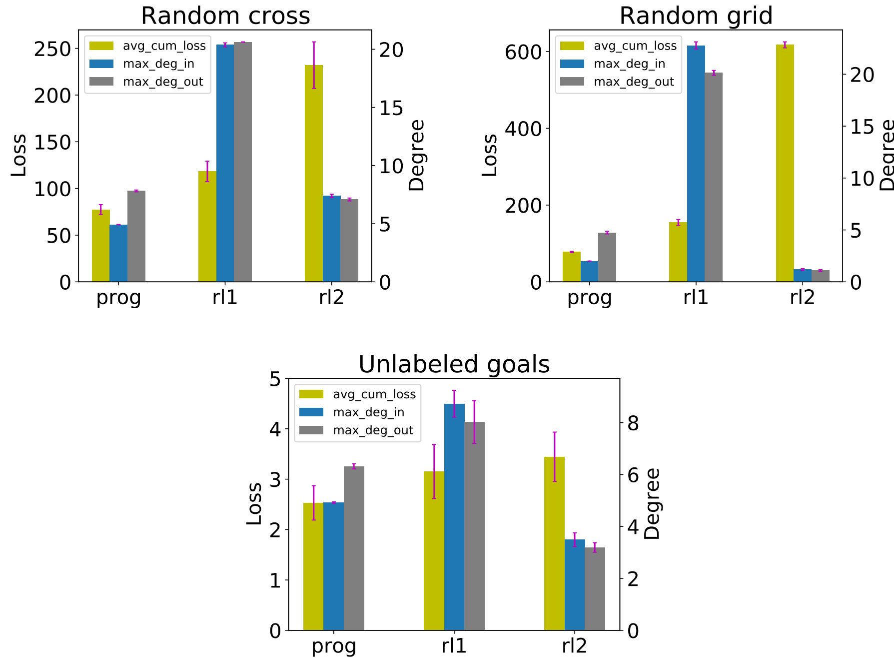


Figure 7-4: Comparing attention program with a RL policy that treats communications as actions. RL1 and RL2 correspond to two different hyper-parameters in the policy gradient algorithm that achieved lowest loss and lowest communication degree (respectively).

Comparison to Communication Decisions as Actions

The multi-agent communication problem can be formulated as an MDP where decisions about which agents to communicate with are part of the action. We performed additional experiments to compare to this approach. Since the action space now includes discrete actions, we used the policy gradient algorithm to train the policy. We tuned several hyper-parameters including (i) weights for balancing the reward term with the communication cost, (ii) whether to use a shaped reward function, and (iii) whether to initialize the policy with the pre-trained transformer policy.

Results are shown in Figure 7-4. Here, `rl1` is a baseline policy that achieved the lowest loss across all hyper-parameters we tried; however, this policy has a very high communication degree. In addition, `rl2` is a policy with lowest communication degree; however, this policy has very high loss.

As can be seen, our approach performs significantly better than the baseline. We

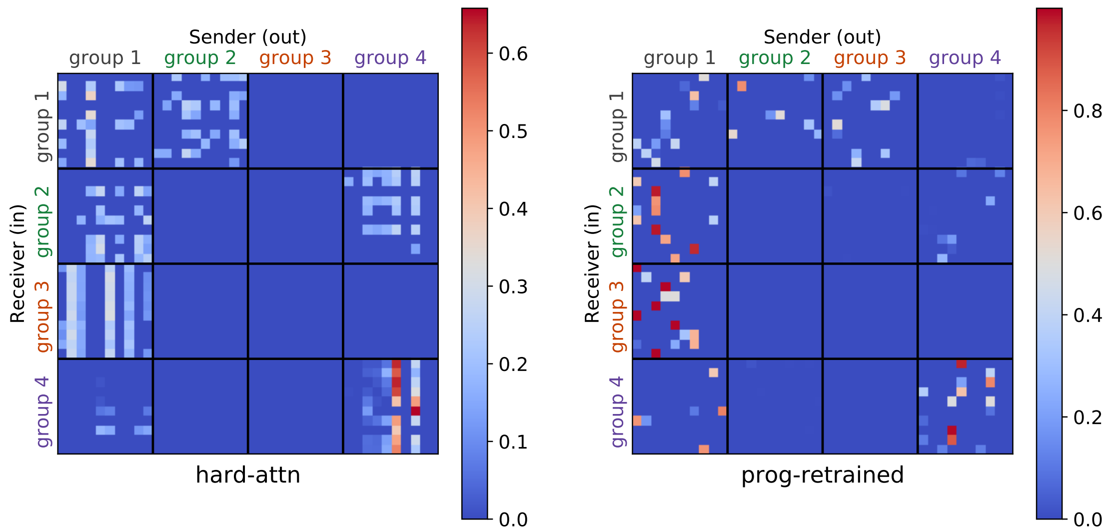


Figure 7-5: Attention weights for `hard-attn` and `prog-retrained` at a single step near the start of a rollout, computed by the agent along the y-axis for the message from the agent along the x-axis.

believe this is due to the combinatorial blowup in the action space—i.e., there is a binary communication decision for each pair of agents, so the number of communication actions is 2^{N-1} per agent and $2^{N(N-1)}$ for all agents (where N is the number of agents). Our approach addresses this challenge by using the transformer as a teacher.

7.7.2 Combinatorial Optimization—Reducing the Communication Degree

From Figure 7-2, it is clear that our approach is the best at optimizing the combinatorial objective. All other approaches (except possibly `fixed`) have higher communication degrees.

Figure 7-5 shows the attention maps of `hard-attn` and `prog-retrained` for the `random-cross` task at a single step. Agents using `hard-attn` often attend to messages from a small subset of agents; thus, even if the maximum in-degree is low, the maximum out-degree is high—i.e., there are a few agents that must send messages to many other agents. In contrast, `prog-retrained` uses randomness to distribute communication across agents.

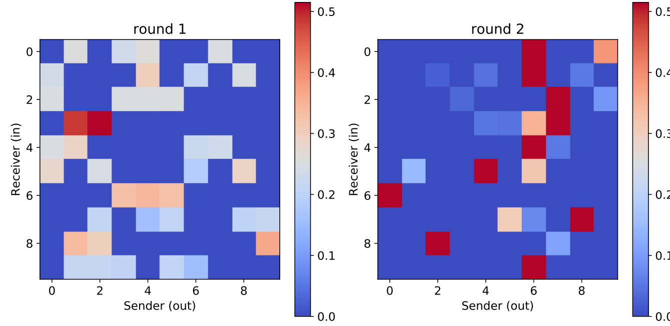


Figure 7-6: Attention maps of `prog-retrained` for the two rounds of communication for the unlabeled goals task.

Figure 7-6 shows the attention maps for `prog-retrained` for the two rounds of communication for the unlabeled goals task.

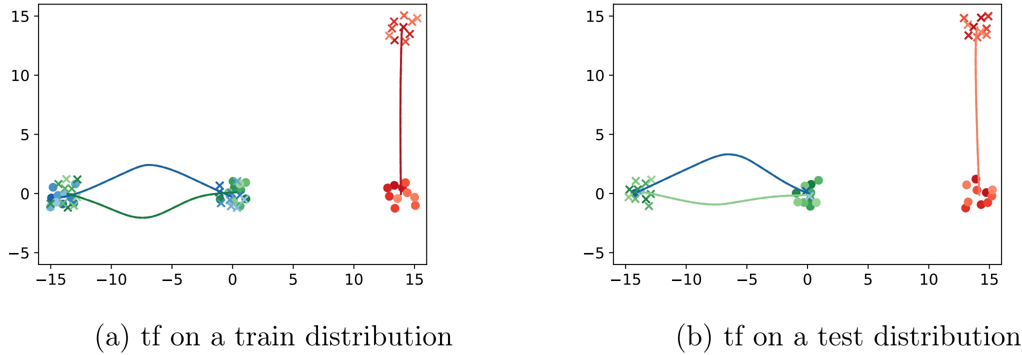
7.7.3 Interpretability

Figure 1-5 visualizes the learned attention program for the `random-grid` task. Here, (a) and (b) visualize the non-deterministic rule for two different configurations. As can be seen, the region from which the rule chooses an agent (depicted in orange) is in the direction of the goal of the agent, presumably to perform longer-term path planning. The deterministic rule (Figure 1-5c) prioritizes choosing a nearby agent, presumably to avoid collisions. Thus, the rules focus on communication with other agents relevant to planning.

7.7.4 Generalization and Robustness

Generalization

To test for generalization, we varied the distribution of agents in the groups for the `random-grid` task. There can now be 2 to 20 agents in each group in this test setup rather than a uniform 10 agents per group. We found that a transformer model trained on the uniform distribution, when tested on this new asymmetric distribution, paid more attention to large groups (maintaining large distance with that group) and less attention to small groups (maintaining a very small distance with that group)



(a) tf on a train distribution

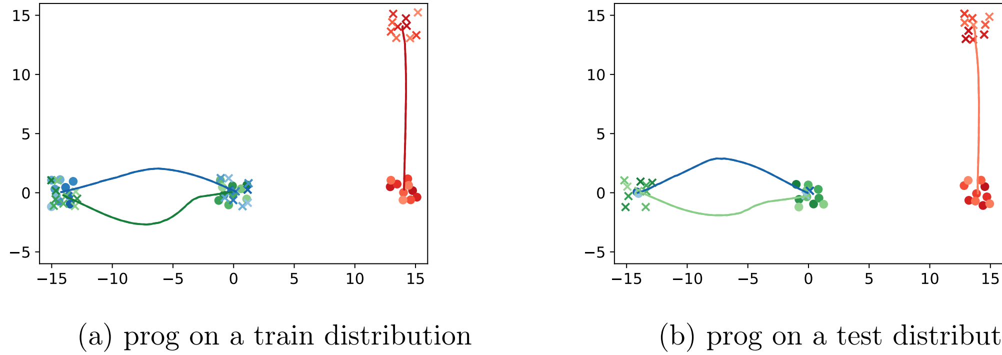
(b) tf on a test distribution

Figure 7-7: For the `random-grid` task, we show how the transformer policy adapts to new settings. On the left, we show the trajectories for a setting from the train distribution, i.e. with an equal number of agents in all groups. On the right, we show the trajectories for a new setting from a different test distribution, i.e. with an asymmetric number of agents in the groups. We see that a transformer policy has an undesirable side-effect; it pays more attention to the group with more agents.

(see Figure 7-7). On the other hand, a neurosymbolic transformer model paid equal attention to all groups (irrespective of their sizes), which is what is needed for this task (see Figure 7-8). We can see why this is the case by looking at the attention program in Figure 1-4d and the visualizations in Figure 1-5; the rule R_1, here, chooses a random agent from the other group irrespective of the size of the group. This experiment is an instance showing that neurosymbolic transformers generalize better than neural transformers.

Case Study with Noisy Communications

We considered a new benchmark based on the random grid task, but the communication link between any pair of agents has a 50% probability of failing. The results are shown in Figure 7-9. As can be seen, the neurosymbolic transformer policy (`prog-retrained`) again has a similar loss as the transformer policy while simultaneously achieving a lower communication degree. Here, the best performing policy has four rules (i.e., $K = 4$), whereas, for the previous random grid task, the attention program only had two rules. Intuitively, each agent attempts to communicate with more of the other agents to compensate for the missing communications.



(a) prog on a train distribution

(b) prog on a test distribution

Figure 7-8: For the `random-grid` task, we show how the neurosymbolic transformer policy adapts to new settings. On the left, we show the trajectories for a setting from the train distribution, i.e. with an equal number of agents in all groups. On the right, we show the trajectories for a new setting from a different test distribution, i.e. with an asymmetric number of agents in the groups. We see that the prog-retrained policy correctly attends to all groups (irrespective of the number of agents in a group).

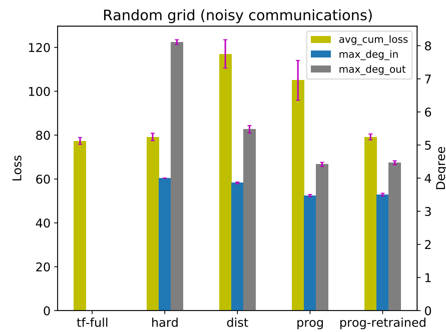


Figure 7-9: Random grid task with noisy communications.

7.8 Discussion

In this case study, we explored neurosymbolic transformer policies for decentralized control of multi-agent systems. Our approach performed as well as state-of-the-art transformer policies while significantly reducing the amount of communication required to achieve complex multi-agent planning goals. There is much room for future work—e.g., exploring other measures of the amount of communication (other combinatorial objectives), better understanding what information is being communicated, and handling environments with more complex observations such as camera images or LIDAR scans. Another direction is to explore more complex multi-agent tasks that require complex/different DSLs for the attention programs (e.g. a DSL that has a consensus module which executes any well known consensus algorithm).

Furthermore, neurosymbolic transformers may have applications in other areas of machine learning such as NLP where transformers are state-of-the-art. In particular, replacing soft attention weights in transformers with programmatic attention rules makes them much easier to interpret and generalize.

Chapter 8

Future Directions

This thesis, so far, showed how neurosymbolic learning could help us get interpretable, generalizable, and robust models in several domains. However, there is much left to be done in the space for neurosymbolic learning. The ultimate goal is to achieve scalability at the level of what deep learning can do currently.

Similar to the organization throughout this thesis, I am organizing the potential future directions in terms of applications, model classes and algorithms.

Applications: Most of the applications explored in this thesis are in robotics. But even within robotics, there are several other applications where the neurosymbolic learning approach applies. Perception and dynamics modelling are examples where I anticipate that inducing program structure in the models can help increase robustness and reduce the amount of data needed for learning. Outside of robotics, some potential domains for neurosymbolic learning techniques are healthcare, computational biology, and finance applications, where it is beneficial to have non-opaque and interpretable models. Image/Scene generation is another domain where compositionality plays a key role, and I believe neurosymbolic approaches can help recover the underlying modular representations for these problems.

Several exciting research directions use machine learning to improve computer systems and computer programming, e.g., big-data based code assistant systems. The idea of incorporating program structure into neural models is can also be applied

to these systems. Neurosymbolic approaches can learn compositional models from code databases that systematically generalize to coding problems in both the same domain and other similar domains where the data might be limited. Thus, closing the loop by using programming to help machine learning to help programming itself.

Model Architectures: Potential future works involve exploring other kinds of neurosymbolic architectures for the models. So far, my model classes only involve parametric programs with loops, conditionals, and simple list operations. However, programs are rich; they can have functions and data structures. One potentially future neurosymbolic architecture is to include programs that can manipulate complex data structures and has functions. Functions are needed for modularity, and data structures are instrumental in efficiently organizing the internal memory of a model; these qualities, in turn, can lead to better generalization. Here, the fundamental challenge would be to develop an over-parameterized representation (the teacher/oracle) that can meaningfully represent these complex data structures and is also amenable to gradient-based approaches.

Another potential future direction is enabling model classes with both neural components and program components, where the programs capture the structure and the logical part of the model, and the neural networks capture the high-dimensionality of the real world. There has been some prior work in this space, but the NNs and the programs have naturally separate specs in these domains. That may not be the case in other applications. For example, in the control problem in Figure 1-1, the input state of the car includes its geometrical positions and velocities. However, to learn a control policy that takes more complex inputs such as camera images or LIDAR images, we want a neural network to handle the perception part but still have a program component to represent the high-level logic.

Algorithms: As we try out more applications and architectures, we will need more techniques to jointly reason about the symbolic and continuous components. An interesting direction in this space is to dynamically learn the right teacher/oracle

such that the teacher can be optimized easily and the teacher is able to provide the right level of guidance to the student. An ultimate goal for neurosymbolic learning is to have an analog of SGD (stochastic gradient descent), ADAM, or other optimization algorithms, i.e. we need one (or a small set of) general purpose algorithm(s) that can handle various neurosymbolic architectures/applications.

Another benefit of learning symbolic models is that we can leverage the fact that formal verification methods are more mature at handling structured programs rather than neural networks. Although we could use some off-the-shelf solvers to verify some of our learned models, it is beneficial to develop more targeted algorithms to formally prove that the neurosymbolic models learned for an intelligent system are safe, robust, and fair.

Finally, both machine-learning and program synthesis have advanced significantly in recent years. For example, machine learning has shifted more towards meta-learning, library learning, and artificial general intelligence. Similarly, there are new neural-based program synthesis techniques and techniques that can learn the DSL from a corpus of tasks. Future neurosymbolic learning algorithms should make use of these latest techniques.

And that's the end of this thesis! I hope I have convinced the readers that neurosymbolic learning with all its benefits can contribute to the world of robust and reliable intelligent systems.

Bibliography

[1] Mohamadreza Ahmadi, Ufuk Topcu, and Clarence Rowley. Control-oriented learning of lagrangian and hamiltonian systems. In *2018 Annual American Control Conference (ACC)*, pages 520–525. IEEE, 2018.

[2] Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 242–252, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[4] Rajeev Alur, Costas Courcoubetis, Thomas A Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid systems*, pages 209–229. Springer, 1992.

[5] Michael Bain and Claude Sammut. A framework for behavioural cloning. In *Machine Intelligence 15*, pages 103–129, 1995.

[6] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*, 2017.

[7] Mislav Balunovic and Martin Vechev. Adversarial training and provable defenses: Bridging the gap. In *International Conference on Learning Representations*, 2019.

[8] Hamsa Bastani, Kimon Drakopoulos, Vishal Gupta, Jon Vlachogiannis, Christos Hadjicristodoulou, Pagona Lagiou, Gkikas Magiorkinis, Dimitrios Paraskevis, and Sotirios Tsiodras. Deploying an artificial intelligence system for covid-19 testing at the greek border. *Available at SSRN*, 2021.

[9] Osbert Bastani. Sample complexity of estimating the policy gradient for nearly deterministic dynamical systems. In *AISTATS*, 2020.

[10] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. *arXiv preprint arXiv:1805.08328*, 2018.

[11] David Bau, Bolei Zhou, Aditya Khosla, Aude Oliva, and Antonio Torralba. Network dissection: Quantifying interpretability of deep visual representations. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6541–6549, 2017.

[12] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016.

[13] Pavol Bielik, Veselin Raychev, and Martin Vechev. Program synthesis for character level language modeling. In *ICLR*, 2017.

[14] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[15] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. *Classification and regression trees*. Routledge, 2017.

[16] Thais Campos, Jeevana Priya Inala, Armando Solar-Lezama, and Hadas Kress-Gazit. Task-based design of ad-hoc modular manipulators. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 6058–6064. IEEE, 2019.

[17] Luca Carlone and Giuseppe C. Calafiore. Convex relaxations for pose graph optimization with outliers. *IEEE Robotics and Automation Letters*, 3(2):1160–1167, 2018.

[18] Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. Web question answering with neurosymbolic program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 328–343, 2021.

[19] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. Program synthesis using deduction-guided reinforcement learning. In *International Conference on Computer Aided Verification*, pages 587–610. Springer, 2020.

[20] Steve Collins, Andy Ruina, Russ Tedrake, and Martijn Wisse. Efficient bipedal robots based on passive-dynamic walkers. *Science*, 307(5712):1082–1085, 2005.

[21] Nilesh Dalvi, Pedro Domingos, Sumit Sanghai, and Deepak Verma. Adversarial classification. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 99–108, 2004.

[22] Abhishek Das, Théophile Gervet, Joshua Romoff, Dhruv Batra, Devi Parikh, Michael Rabbat, and Joelle Pineau. Tarmac: Targeted multi-agent communication. In *ICML*, 2019.

[23] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[24] Marc Deisenroth and Carl E Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pages 465–472, 2011.

[25] Alexandre Donzé, Bruce Krogh, and Akshay Rajhans. Parameter synthesis for hybrid systems with an application to simulink models. In *International workshop on hybrid systems: Computation and control*, pages 165–179. Springer, 2009.

[26] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[27] Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. Inversecsg: Automatic conversion of 3d models to csg trees. *ACM Transactions on Graphics (TOG)*, 37(6):1–16, 2018.

[28] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In *Advances in neural information processing systems*, pages 6059–6068, 2018.

[29] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Unsupervised learning by program synthesis. 2015.

[30] Kevin M Ellis, Lucas E Morales, Mathias Sablé-Meyer, Armando Solar Lezama, and Joshua B Tenenbaum. Library learning for neurally-guided bayesian program induction. 2018.

[31] Kevin M Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Joshua Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. 2019.

[32] Dumitru Erhan, Yoshua Bengio, Aaron Courville, and Pascal Vincent. Visualizing higher-layer features of a deep network. *University of Montreal*, 1341(3):1, 2009.

[33] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices*, 50(6):229–239, 2015.

[34] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International*

Conference on Machine Learning-Volume 70, pages 1126–1135. JMLR. org, 2017.

[35] Jakob Foerster, Ioannis Alexandros Assael, Nando De Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. In *Advances in neural information processing systems*, pages 2137–2145, 2016.

[36] Jakob N Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[37] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2018.

[38] Amir Globerson and Sam Roweis. Nightmare at test time: robust learning by feature deletion. In *Proceedings of the 23rd international conference on Machine learning*, pages 353–360, 2006.

[39] Robert L Grossman, Anil Nerode, Anders P Ravn, and Hans Rischel. *Hybrid systems*, volume 736. Springer, 1993.

[40] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[41] Sumit Gulwani. Programming by examples. *Dependable Software Systems Engineering*, 45(137):3–15, 2016.

[42] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[43] W Keith Hastings. Monte carlo sampling methods using markov chains and their applications. 1970.

[44] Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455*, 2015.

[45] Thomas A Henzinger. The theory of hybrid automata. In *Verification of digital and hybrid systems*, pages 265–292. Springer, 2000.

[46] Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. Synthesizing programmatic policies that inductively generalize. In *International Conference on Learning Representations*, 2020.

[47] Jeevana Priya Inala, Yichen Yang, James Paulos, Yewen Pu, Osbert Bastani, Vijay Kumar, Martin Rinard, and Armando Solar-Lezama. Neurosymbolic transformers for multi-agent communication. *Advances in Neural Information Processing Systems*, 33, 2020.

[48] Jeevana Priya Inala, Yichen Yang, James Paulos, Yewen Pu, Osbert Bastani, Vijay Kumar, Martin Rinard, and Armando Solar-Lezama. Neurosymbolic transformers for multi-agent communication. In *Neural Information Processing Systems*, 2020.

[49] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Synthesizing switching logic for safety and dwell-time requirements. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, pages 22–31, 2010.

[50] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

[51] Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*, pages 443–452. Springer, 2019.

[52] Arbaaz Khan, Ekaterina Tolstaya, Alejandro Ribeiro, and Vijay Kumar. Graph policy gradients for large scale robot control. In *CoRL*, 2019.

[53] Arbaaz Khan, Chi Zhang, Shuo Li, Jiayue Wu, Brent Schlotfeldt, Sarah Y Tang, Alejandro Ribeiro, Osbert Bastani, and Vijay Kumar. Learning safe unlabeled multi-robot planning with motion constraints. In *IROS*, 2019.

[54] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[55] Soonho Kong, Sicun Gao, Wei Chen, and Edmund Clarke. dreach: δ-reachability analysis for hybrid systems. In *International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems*, pages 200–205. Springer, 2015.

[56] Tim Kraska, Mohammad Alizadeh, Alex Beutel, H Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. In *CIDR*, 2019.

[57] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[58] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[59] Sergey Levine and Vladlen Koltun. Guided policy search. In *International Conference on Machine Learning*, pages 1–9, 2013.

[60] 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. *arXiv preprint arXiv:1509.02971*, 2015.

[61] Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pages 157–163. Elsevier, 1994.

[62] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in neural information processing systems*, pages 6379–6390, 2017.

[63] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks, 2019.

[64] Aravindh Mahendran and Andrea Vedaldi. Understanding deep image representations by inverting them. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5188–5196, 2015.

[65] Shie Mannor, Reuven Y Rubinstein, and Yohai Gat. The cross entropy method for fast policy search. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 512–519, 2003.

[66] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[67] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[68] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In *International Conference on Machine Learning*, pages 4861–4870. PMLR, 2019.

[69] Maxwell I Nye, Armando Solar-Lezama, Joshua B Tenenbaum, and Brenden M Lake. Learning compositional rules via neural program synthesis. *arXiv preprint arXiv:2003.05562*, 2020.

[70] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. Feature visualization. *Distill*, 2(11):e7, 2017.

[71] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, pages 619–630, 2015.

[72] Charles Packer, Katelyn Gao, Jernej Kos, Philipp Krähenbühl, Vladlen Koltun, and Dawn Song. Assessing generalization in deep reinforcement learning. *arXiv preprint arXiv:1810.12282*, 2018.

[73] Simone Paoletti, Aleksandar Lj Juloski, Giancarlo Ferrari-Trecate, and René Vidal. Identification of hybrid systems a tutorial. *European journal of control*, 13(2-3):242–260, 2007.

[74] James Paulos, Steven W Chen, Daigo Shishika, and Vijay Kumar. Decentralization of multiagent policies by learning what to communicate. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 7990–7996. IEEE, 2019.

[75] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[76] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, 2015.

[77] Martin L Puterman. Markov decision processes. *Handbooks in operations research and management science*, 2:331–434, 1990.

[78] Aniruddh Raghu, Matthieu Komorowski, Leo Anthony Celi, Peter Szolovits, and Marzyeh Ghassemi. Continuous state-space models for optimal sepsis treatment: a deep reinforcement learning approach. In *Machine Learning for Healthcare Conference*, pages 147–163. PMLR, 2017.

[79] Aravind Rajeswaran, Kendall Lowrey, Emanuel V Todorov, and Sham M Kakade. Towards generalization and simplicity in continuous control. In *Advances in Neural Information Processing Systems*, pages 6550–6561, 2017.

[80] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

[81] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[82] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ASPLOS*, pages 305–316, 2013.

[83] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, 2013.

[84] Ludwig Schmidt, Shibani Santurkar, Dimitris Tsipras, Kunal Talwar, and Aleksander Mądry. Adversarially robust generalization requires more data, 2018.

[85] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[86] Ameesh Shah, Eric Zhan, Jennifer J Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. Learning differentiable programs with admissible neural heuristics. *arXiv preprint arXiv:2007.12101*, 2020.

[87] Daigo Shishika, James Paulos, and Vijay Kumar. Cooperative team strategies for multi-player perimeter-defense games. *IEEE Robotics and Automation Letters*, 5(2):2738–2745, 2020.

[88] 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. *nature*, 529(7587):484–489, 2016.

[89] Amanpreet Singh, Tushar Jain, and Sainbayar Sukhbaatar. Learning when to communicate at scale in multiagent cooperative and competitive tasks. *arXiv preprint arXiv:1812.09755*, 2018.

[90] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *FSE*, pages 289–299, 2011.

[91] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5):475–495, 2013.

[92] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[93] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[94] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.

[95] Ekaterina Tolstaya, Fernando Gama, James Paulos, George Pappas, Vijay Kumar, and Alejandro Ribeiro. Learning decentralized controllers for robot swarms with graph neural networks. In *CoRL*, 2019.

[96] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: specifying protocols with concolic snippets. In *PLDI*, pages 287–296, 2013.

[97] Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. Houdini: lifelong learning as program synthesis. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 8701–8712, 2018.

[98] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[99] Abhinav Verma, Hoang Minh Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected policy gradient for programmatic reinforcement learning. *CoRR*, abs/1907.05431, 2019.

[100] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, pages 5045–5054. PMLR, 2018.

[101] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *arXiv preprint arXiv:1911.04942*, 2019.

[102] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466, 2017.

[103] Fulton Wang and Cynthia Rudin. Falling rule lists. In *Artificial Intelligence and Statistics*, pages 1013–1022. PMLR, 2015.

[104] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *PLDI*, pages 508–521, 2016.

[105] Halley Young, Osbert Bastani, and Mayur Naik. Learning neurosymbolic generative models via program synthesis. In *International Conference on Machine Learning*, pages 7144–7153. PMLR, 2019.

[106] Xin Zhang, Armando Solar-Lezama, and Rishabh Singh. Interpreting neural network judgments via minimal, stable, and symbolic corrections. *arXiv preprint arXiv:1802.07384*, 2018.

[107] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. An inductive synthesis framework for verifiable reinforcement learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 686–701. ACM, 2019.