New Algorithms for Programmatic Deep Learning with Applications to Behavior Modeling

Thesis by

Eric Zhan

In Partial Fulfillment of the Requirements for the
Degree of
Doctor of Philosophy

CALIFORNIA INSTITUTE OF TECHNOLOGY
Pasadena, California

2022
Defended November 16, 2021

Eric Zhan
ORCID: 0000-0003-0521-5329

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my Ph.D. advisor, Yisong Yue. I remember during our first one-on-one meeting I expressed all my worries about starting my Ph.D., yet you reassured me to "trust the process". I think those words also meant for me to put my trust in you, and I'm glad I did. Looking back at my journey, I'm amazed and proud of what I've achieved—from my first paper submission, to my first paper rejection, my second paper rejection, and then my third (all part of the process), to my first conference publication, my first research internship, and eventually to this thesis—all this would not have been possible without your guidance. Your patience, support, and kindness over these last 5 years have been invaluable to me. Thank you for being the best advisor I could have hoped for.

Thank you to my committee members, Richard Murray, Pietro Perona, and Swarat Chaudhuri, for taking the time to provide feedback and help me improve my research.

I had the privilege to collaborate with many talented researchers during my Ph.D. (in no particular order): Stephan Zheng, Long Sha, Patrick Lucey, Yukai Liu, Rose Yu, Albert Tseng, Adith Swaminathan, Matthew Hausknecht, Ameesh Shah, Jennifer Sun, Abhinav Verma, Swarat Chaudhuri, Oliver Stephenson, Tobias Köhne, Brent Cahill, Sang-Ho Yun, Zachary Ross, Mark Simons, Ann Kennedy, David Anderson, Pietro Perona, Andrew Hartnett, and Jagjeet Singh. Thank you all for your hard work. It was truly a pleasure to have met and worked with all of you.

I also partook in two summer research internships while completing my degree. Thank you to Adith Swaminathan and Matthew Hausknecht at Microsoft Research, and Andrew Hartnett and Greydon Foil at Argo AI for being my mentors and giving me a taste of industry research. I learned a lot from both experiences.

Thank you to everyone I met at Caltech and all my friends, including the Yue Crew (amazing to see it grow from 5 members when I started to now 14), my CMS cohort with Sara, Natalie, Riley, and Chen, my officemates Nikola and Gautam, and my awesome roommate Xichen. Thanks for all the great memories over the years.

Last but certainly not least, thank you to my parents, my sister, and my grandparents for their unconditional love and support while I was across the country pursuing my studies. It is only thanks to you that I had this opportunity at all, and I hope you are as happy for me as I am.

ABSTRACT

Raw behavioral data is becoming increasingly more abundant and more easily obtainable in spatiotemporal domains such as sports, video games, navigation & driving, motion capture, and animal science. How can we best use this data to advance their respective domains forward? For instance, researchers for self-driving vehicles would like to identify the key features of the environment state that impact decision-making the most; game developers would like to populate their games with characters that have unique and diverse behaviors to create a more immersive gaming experience; and behavioral neuroscientists would like to uncover the underlying mechanisms that drive learning in animals. Machine learning, the science of developing models and algorithms to identify and leverage patterns in data, is well-equipped to aid in these endeavors. But how do we integrate machine learning with these spatiotemporal domains in a principled way? In this thesis, we develop and introduce new algorithms in *programmatic deep learning* that tackle some of the new challenges encountered in behavior modeling.

Our work in programmatic deep learning comprises two main themes: in the first, we show how to use expert-written programs as sources of weak labels in domains where manually-annotated expert labels are scarce; in the second, we explore programs as a flexible function class with human-interpretable structure and show how to learn them via neurosymbolic program learning. Augmenting deep learning with programmatic structure allows domain experts to easily incorporate domain knowledge into machine learning models; we show that this results in significant improvements in many behavior modeling applications like imitation learning, controllable generation, counterfactual analysis, and unsupervised clustering.

PUBLISHED CONTENT AND CONTRIBUTIONS

Sun, Jennifer J et al. (2021). "Task programming: Learning data efficient behavior representations". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2876–2885. URL: https://openaccess.thecvf.com/content/CVPR2021/html/Sun_Task_Programming_Learning_Data_Efficient_Behavior_Representations_CVPR_2021_paper.html.
E.Z. participated in the formulation of the method, conducted experiments and analyzed results, and participated in the writing of the manuscript.

Zhan, Eric, Jennifer J Sun, et al. (2021). "Unsupervised Learning of Neurosymbolic Encoders". In: *arXiv preprint arXiv:2107.13132*. URL: https://arxiv.org/abs/2107.13132.
E.Z. participated in the conception of the project, formulated and implemented the method, conducted experiments and analyzed results, and participated in the writing of the manuscript.

Shah, Ameesh et al. (2020). "Learning Differentiable Programs with Admissible Neural Heuristics". In: *Advances in Neural Information Processing Systems*. Vol. 33, pp. 4940–4952. URL: https://proceedings.neurips.cc/paper/2020/hash/342285bb2a8cadef22f667eeb6a63732-Abstract.html.
E.Z. formulated and implemented the method, conducted experiments and analyzed results, and participated in the writing of the manuscript.

Zhan, Eric, Albert Tseng, et al. (2020). "Learning Calibratable Policies using Programmatic Style-Consistency". In: *International Conference on Machine Learning*. PMLR, pp. 11001–11011. URL: https://proceedings.mlr.press/v119/zhan20a.html.
E.Z. participated in the conception of the project, formulated and implemented the method, conducted experiments and analyzed results, and participated in the writing of the manuscript.

Liu, Yukai et al. (2019). "Naomi: Non-autoregressive multiresolution sequence imputations". In: *Advances in Neural Information Processing Systems*. Vol. 32. URL: https://papers.nips.cc/paper/2019/hash/50c1f44e426560f3f2cdcb3e19e39903-Abstract.html.
E.Z. participated in the formulation of the method, conducted experiments and analyzed results, and participated in the writing of the manuscript.

Zhan, Eric, Stephan Zheng, et al. (2019). "Generating Multi-Agent Trajectories using Programmatic Weak Supervision". In: *International Conference on Learning Representation*. URL: https://openreview.net/forum?id=rkxw-hAcFQ.
E.Z. participated in the conception of the project, formulated and implemented the method, conducted experiments and analyzed results, and participated in the writing of the manuscript.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Number *Page*

LIST OF TABLES

Chapter 1

INTRODUCTION

Raw behavioral data is becoming increasingly more abundant and more easily obtainable in spatiotemporal domains such as sports, video games, navigation & driving, motion capture, and animal science. How can we best use this data to advance their respective domains forward? For instance, researchers for self-driving vehicles would like to identify the key features of the environment state that impacts decision-making the most; game developers would like to populate their games with characters that have unique and diverse behaviors to create a more immersive gaming experience; and behavioral neuroscientists would like to uncover the underlying mechanisms that drive learning in animals. Machine learning, the science of developing models and algorithms to identify and leverage patterns in data, is well-equipped to aid in these endeavors. But how do we integrate machine learning with these spatiotemporal domains in a principled way? In this thesis, we develop and introduce new algorithms in *programmatic deep learning* that tackle some of the new challenges encountered in behavior modeling.

1.1 Motivation for Programmatic Deep Learning

Machine learning is the science of developing models and algorithms to identify and leverage patterns in data. The widespread availability of large-scale datasets and the steady increase in compute power has enabled machine learning research to flourish in recent years, yielding many successful applications in domains like computer vision, natural language processing, and autonomous systems. Many successful applications leverage deep learning, a subfield of machine learning that uses deep neural networks as the primary model choice. Neural networks are advantageous because they are universal function approximators that can theoretically model any complex function, but in turn also require a lot of data to generalize beyond the training set. As a result, deep learning models are often coupled with some form of inductive bias to leverage structure in data that can potentially make the task easier to learn. For example, convolutional neural networks work well for image domains because they leverage translation invariance, whereas graph neural networks are designed to leverage permutation invariance for relational data.

This thesis focuses on developing machine learning algorithms for spatiotemporal

domains, particularly behavior modeling settings where agents interact in an environment and possibly with each other. Figure 1.1 depicts two such domains, namely the tracking of laboratory mice in an enclosure, and the tracking of professional basketball players from real NBA games.

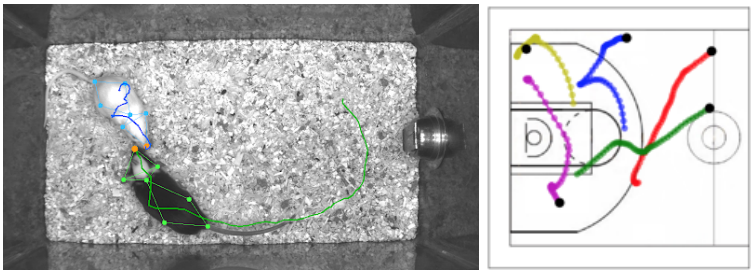


Figure 1.1: Two examples of tracking domains for behavior modeling. Left: two mice interacting in a rectangular enclosure. Dots represent key body parts of each mouse (e.g. nose, ears, tail). Right: the trajectories of five professional basketball players over 8 seconds, starting from the black dots. The ball is not shown.

Deep learning in sequential domains typically use recurrent neural networks (RNN) to leverage the temporal structure of the data by maintaining an internal state/memory for capturing long-term dependencies (Hochreiter and Schmidhuber, 1997; Cho et al., 2014). While RNNs generally work well for sequential tasks like time series forecasting and speech recognition, behavior modeling settings have several properties that introduce additional challenges:

- Agent behavior can be highly coordinated. For example in team basketball in Figure 1.1 Right, players are moving together to execute a specific strategy rather than moving independently. The difficulty of behavior modeling scales exponentially with respect to the number of agents in multi-agent settings.
- Agent behavior can be non-deterministic and multimodal. For example in animal tracking, upon reaching a wall a mouse might turn left or right with nonzero probability. Another example is in competitive sports, where it can be beneficial to be unpredictable in order to fool the opposing team. A challenge in behavior modeling is capturing the entire distribution of possible behaviors.
- Behavioral data is often collected from observing different agents that can each exhibit distinct behavior styles and differing intrinsic goals, e.g., aggressive vs. risk-averse drivers, male vs. female mice, etc. This can be be challenging for some behavior modeling applications, like controllable generation and counterfactual reasoning.

- The amount of raw behavioral data is very large, but expert annotations can be very expensive to acquire. Learning algorithms most likely have to be entirely unsupervised or semi-supervised, or take label-efficiency into careful consideration.

- Model interpretability is important for many domain experts working on downstream tasks in behavior modeling. For example, interpretability is key in behavioral neuroscience for understanding what leads to lifelong learning in animals. Additionally, understanding why a model made a prediction can assist in debugging and lead to future improvements, e.g., *"Why did the model predict that the vehicle should slow down?"*. Deep learning is a powerful tool, but it is also notorious for being very difficult to interpret.

In light of these challenges, we develop new algorithms for behavior modeling that integrate *programmatic* structure with deep learning. We refer to this as **programmatic deep learning**, which encompasses two main themes in this thesis: 1) using expert-written programs as weak label sources, and 2) neurosymbolic program learning. Programs offer an easy way for domain experts to incorporate domain knowledge and specify structure that introduces a strong yet flexible inductive bias. We show that programmatic structure can lead to many improvements in behavior modeling applications like imitation learning, controllable generation, counterfactual analysis, and unsupervised clustering

Theme 1: programs as weak label sources

Conventionally, inspecting data examples and providing annotations is how a domain expert supplements data with domain knowledge. However, the amount of raw behavioral data is often very large, which means it can be very expensive to acquire expert labels for an entire dataset. Fortunately, although expert labels can be quite scarce, expert programs written by domain experts to compute heuristics can be very plentiful. For example, behavioral neuroscientists have programs to characterize the social behavior of mice (Segalin et al., 2020), and sport analysts have programs to automatically recognize specific maneuvers like on-ball screens in basketball (McQueen, Wiens, and Guttag, 2014). Compared to manually labeling each data example, expert programs offer a computationally inexpensive solution for domain experts to incorporate domain knowledge and is a much more efficient use of a domain expert's time. The trade-off, however, is that the resulting labels are often noisy. Using expert programs as weak label sources in place of ground-truth labels

was first explored in the data programming framework for binary classification (Ratner et al., 2016). In this thesis, we show how to use expert programs to address some of the challenges encountered in behavior modeling, like coordinated multi-agent behavior and controllable generation of behaviors.

Theme 2: neurosymbolic program learning

Inductive program synthesis refers to the task of generating a program in some domain-specific language (DSL) that matches pairs of input/output examples. Traditional approaches formulate this problem as search over the combinatorial space of possible programs, which often does not scale as well as deep learning to high-dimensional domains that can require more complex program structures. However, program learning does boast several advantages:

- Programs are much more interpretable. For example, one interpretation of the program in Figure 1.2 is that it computes the Euclidean distance between the players and the ball to identify the ballhandler (player with possession of the ball).
- The DSL serves as an inductive bias for the learning algorithm. Domain experts can encode domain knowledge via the DSL library functions, which greatly impacts the final result (e.g. program in Figure 1.2). In addition, this programmatic inductive bias can sometimes lead to better generalization to new inputs (Verma, Murali, et al., 2018; Verma, Le, et al., 2019).
- Programs are highly compositional, which is ideal for downstream tasks like transferable lifelong learning (Valkov et al., 2018) and library learning (K. M. Ellis et al., 2018; K. Ellis et al., 2020).

Neurosymbolic program learning studies how we can combine the best of both worlds: the symbolic structure of programs with the scalability of neural networks. Neurosymbolic programs are often defined to have both symbolic and neural components (Valkov et al., 2018). The challenge lies in how to efficiently solve for both the discrete optimization problem of finding the optimal program structure with the continuous optimization problem of learning optimal neural network parameters. In this thesis, we develop an efficient algorithm for learning differentiable neurosymbolic programs and demonstrate their effectiveness for behavior classification and unsupervised clustering in behavior modeling.

```
map(
  multiply(
    add(OffenseAffine(s_t), BallAffine(s_t)),
    add(OffenseAffine(s_t), BallAffine(s_t))
  ),
τ)
```

Figure 1.2: Example program for identifying the ballhander in basketball. **map** applies the function in the first argument to every state $\mathbf{s}_t$ in trajectory τ. `OffenseAffine` and `BallAffine` are DSL library functions that extract the xy-coordinates of the offensive players and the ball from state $\mathbf{s}_t$, before applying a parameterized affine transformation. The interpretation is that the program looks to be computing the Euclidean distance between players and the ball.

In the following section, we first introduce behavior modeling more formally, and then give brief overviews of machine learning problems in behavior modeling for which programmatic deep learning can be practical.

1.2 Machine Learning for Behavior Modeling

Behavior Modeling as a Markov Decision Process

Behavior modeling settings are commonly modeled as a Markov decision process (MDP), which is formally described as a tuple $(\mathcal{S}, \mathcal{A}, P, R)$. At every timestep t, an agent observes the state of the environment $\mathbf{s}_t \in \mathcal{S}$ and uses its policy π to select and perform an action $\mathbf{a}_t \in \mathcal{A}$. The environment then transitions to the next state via the dynamics $P(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ and the agent receives a scalar reward r_t according to reward function $R(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$.

For behavior modeling settings covered in this thesis, we additionally assume that states are fully-observable and that state and action spaces $\mathcal{S}$ and $\mathcal{A}$ are finite-dimensional. Furthermore, we assume that agents are behaving optimally to maximize their expected cumulative reward. In many behavior modeling settings, however, the exact reward function that agents are optimizing for is not well-defined and also cannot be easily evaluated (e.g., what is the reward function that guides the behavior of mice?). Therefore, explicit rewards r_t are usually not provided as part of behavioral datasets.

Behavioral data is commonly represented as trajectories $\tau = (\mathbf{s}_1, \mathbf{a}_1, \mathbf{s}_2, \mathbf{a}_2, \dots)$, which can also be interpreted as sequences of state-action pairs. In settings where the dynamics are known and deterministic, trajectories can sometimes be reduced to

simply sequences of states $\tau = (\mathbf{s}_1, \mathbf{s}_2, \dots)$. For example, in tracking domains where states and actions are agent positions and velocities respectively, the dynamics are linear, $\mathbf{s}_{t+1} = \mathbf{s}_t + \mathbf{a}_t$, which also means that actions can be recovered by computing the difference in states.

Imitation Learning

The goal of imitation learning is to recover the optimal policy using the collected demonstration data. The simplest approach is *behavioral cloning*, which treats state-action pairs as i.i.d. examples and learns a policy via supervised learning (Pomerleau, 1989). While behavioral cloning can work for simple applications with sufficient data coverage, it can fail to generalize in test scenarios because actions taken by an agent in an MDP will induce future states via the environment dynamics, which breaks the aforementioned i.i.d. assumption. Consequently, errors made by the agent can compound and lead the agent to drift towards states unseen during training, potentially resulting in catastrophic failures. To address this shortcoming, interactive imitation learning algorithms introduce multiple rounds of supervised learning, policy rollout, and oracle feedback for new data to iteratively improve the policy (Ross, Gordon, and D. Bagnell, 2011). Inverse reinforcement learning is another class of algorithms that aims to reverse-engineer the unknown reward function from data and feed it into reinforcement learning algorithms to learn the policy (Abbeel and Ng, 2004; Ziebart et al., 2008).

In particular in behavior modeling settings, agent behavior can be inherently non-deterministic and multimodal. For example, a basketball player can dribble around the left or right of a defender; both possibilities are realistic, but the mean action, i.e. dribbling into the defender, would be unnatural in this scenario. Thus, there has been recent interest in learning stochastic policies that capture entire distributions of possible actions (Hrolenok, Boots, and Balch, 2017). Initial approaches model an explicit distribution over actions in the final layer of a deep model (Zheng, Yue, and Lucey, 2016; Eyjolfsdottir, K. Branson, et al., 2017). More recent approaches leverage deep generative modeling to learn transformations from simple to more complex distributions over actions using either latent variable modeling (Johnson et al., 2016; Chung et al., 2015) and/or adversarial training (Ho and Ermon, 2016; Wang et al., 2017).

Imitation learning of behaviors continues to be challenging in terms of multimodality, scale, and long-term consistency. In Chapter 2, we tackle imitation learning of

coordinated multi-agent behavior over long time horizons.

Controllable Generation of Behavior Styles

A common feature of behavioral datasets is that trajectories are often collected from different individual agents, each potentially exhibiting distinct behavior styles, e.g., new/experienced drivers, aggressive/passive basketball players, etc. All styles are representative of realistic behavior; there isn't necessarily one style that is most "optimal" compared to the rest. Given a diverse range of behavior styles, can we learn policies that can be controlled to imitate different behavior styles? Since behavior styles are rarely labeled, imitation learning of diverse behaviors has relied on unsupervised methods to infer latent codes that capture behavior styles while jointly learning a policy conditioned on these codes to generate the corresponding behavior style (Wang et al., 2017).

A crucial component of learning controllable style-conditioned policies is ensuring that behavior styles are *consistently* generated when desired. Typical strategies aim to induce a tight correlation between latent codes and policy rollouts, such as by maximizing the mutual information between the two (Yunzhu Li, Song, and Ermon, 2017; Hausman et al., 2017; Sharma et al., 2020), which is inspired by the recent success of conditional generation in image domains (X. Chen, Duan, et al., 2016; Creswell et al., 2017). Our approach in Chapter 3 formally introduces *style-consistency* as a learning objective that can be integrated into existing imitation learning algorithms.

Controllable style-consistent policies are pivotal for empowering many downstream tasks like realistic simulation, virtual agent design, long-term planning, and counterfactual behavior reasoning.

Representation Learning

Many successful machine learning algorithms for behavior modeling tasks rely on a good set of input features. For example, it is common in behavior classification to compute spatiotemporal trajectory features at varying temporal resolutions (Burgos-Artizzu et al., 2012; Eyjolfsdottir, S. Branson, et al., 2014; Hong et al., 2015). This can be very time-consuming for the domain experts, potentially requiring many iterations between feature engineering and model training.

Instead, representation learning aims to automatically extract features that are effective for downstream tasks. Typical approaches use encoder-decoder frameworks

to map raw trajectory data to low-dimensional vector representations (Luxem et al., 2020). Ultimately, the features that are encoded and the information retained in the representation depend on the decoding task used to train the model. For example, autoencoding (reconstructing the input from the representation) has worked well for learning representations for visual data (Vincent et al., 2010) and language modeling (Radford et al., 2018). Self-supervision and contrastive learning introduce decoding tasks that use data augmentations as learning signals for learning representations when ground-truth labels are unavailable (Schroff, Kalenichenko, and Philbin, 2015; Oord, Yazhe Li, and Vinyals, 2018; T. Chen et al., 2020).

Relatively speaking, decoding tasks for learning representations in behavior modeling are underexplored, e.g., what are the natural data augmentations for trajectories that one should use for self-supervision? Ideally, a method for learning good representations would greatly reduce the total amount of domain expert effort required for behavior modeling.

Interpretable Clustering of Behaviors

A popular research direction in behavior modeling is discovering and identifying behavior motifs, i.e., semantically meaningful clusters of behaviors (Berman et al., 2014; Luxem et al., 2020). In general, there are two main strategies.

The first approach builds upon representation learning by applying a clustering algorithm (e.g., k-means clustering) on learned representations. Many such approaches often include additional supervision in the representation learning step to encourage similar trajectories to be mapped closer together in the representation space. The exact notion of similarity is usually implicitly embedded in the learning objective for representation learning, such as with triplet loss (Schroff, Kalenichenko, and Philbin, 2015).

The second approach aims to learn the clusters directly by mapping trajectories down to one of several classes. This is usually done with a latent variable model, either with discrete latent variables (Oord, Vinyals, and Kavukcuoglu, 2017; Dupont, 2018) or with a mixture model prior (Dilokthanakul et al., 2016). However, these models can be tricky to train, as posterior and index collapse are common occurrences (X. Chen, Kingma, et al., 2017; Kaiser et al., 2018).

For both approaches, the interpretability of clusters remains a major challenge. Currently, the only solution is to have domain experts manually inspect the clusters to identify distinguishing characteristics, which can be very difficult and time-

consuming. Our work in neurosymoblic program learning offers another solution by learning *programmatic* representations of data that can be explained with the DSL defined by the domain experts themselves.

1.3 Thesis Structure and Contributions

We introduce new programmatic deep learning algorithms for behavior modeling under our two main themes: Chapters 2 and 3 show how to leverage expert-written programs as weak label sources; Chapters 4 and 5 introduce new algorithms for neurosymbolic program learning. We primarily focus our experiments on tracking domains like sports analytics and animal behavioral science that have known and linear dynamics, and include synthetic experiments where applicable to build intuition for our algorithms. Each chapter also contains a section that details more related work.

In Chapter 2, we study imitation learning in a very challenging multi-agent setting where behaviors are stochastic and highly coordinated. The main difficulty lies in ensuring that agent behavior remains consistent between each other and across long time horizons. Our approach introduces a hierarchical *macro-intent* variable to capture the coordination between agents and encode long-term intentions. Instead of learning macro-intents via unsupervised learning, we use a program written by a domain expert to obtain weak macro-intent labels. We show that this approach is significantly more effective for generating realistic multi-agent behavior, which we verify with a user study and counterfactual analysis experiments.

In Chapter 3, we take a closer look at learning controllable policies. We observe that while our macro-intent variables in Chapter 2 give us better control of our learned policies than previous baselines do, the resulting behavior is only sometimes consistent with the conditioned macro-intent. This motivates our work in formalizing *style-consistency* as a learning objective for learning controllable policies, which we define as the notion that behaviors exhibited from a style-conditioned policy should always match that behavior style. The missing ingredient is how we define behavior styles and measure style-consistency. Our framework uses expert-written programs to produce style labels, which allows us to efficiently check if a style is correctly exhibited. Our solution also allows domain experts specify a behavior style that they would like to control for, as along as the style can be captured with a program.

In Chapter 4, we introduce a new algorithm for learning differentiable neurosymbolic programs. As mentioned previously, neurosymbolic program learning often

boils down to efficiently solving both the discrete optimization problem of finding the optimal program structure and the continuous optimization problem of learning optimal neural network parameters. Our key contribution is framing this optimization problem as search over a weighted graph whose paths encode top-down program derivations. We then use neural networks as continuous relaxations to complete any partial programs in the graph, and show that the resulting training loss of these relaxed partial programs is an approximately admissible heuristic that can guide the graph search. We instantiate our approach, called NEAR for ***Ne**ural **A**dmissible **R**elaxation*, with heuristic-guided search algorithms like A* search to efficiently find programs for behavior classification that yield natural interpretations and achieve competitive accuracy.

In Chapter 5, we present a framework for learning *neurosymbolic encoders* for unsupervised representation learning of behavioral data. Our framework partitions the latent representation into neural and programmatic components and leverages NEAR from Chapter 4 to learn a programmatic encoder for the latter. The final result is a data representation with a clean programmatic interpretation that also significantly outperforms purely neural approaches in extracting semantically meaningful clusters of behaviors. We further showcase the practicality of our framework by integrating our programmatic encoder with task programming, a state-of-the-art self-supervised approach for learning label-efficient representations for behavioral data.

Lastly, we conclude in Chapter 6 with some closing remarks and highlight some exciting directions for future work.

Chapter 2

MULTI-AGENT IMITATION LEARNING WITH PROGRAMMATIC MACRO-INTENTS

The work in this chapter was published in (Zhan et al., 2019). E.Z. participated in the conception of the project, formulated and implemented the method, conducted experiments and analyzed results, and participated in the writing of the manuscript.

Summary

We study the problem of training sequential generative models for capturing coordinated multi-agent trajectory behavior, such as offensive basketball gameplay. When modeling such settings, it is often beneficial to design hierarchical models that can capture long-term coordination using intermediate variables. Furthermore, these intermediate variables should capture interesting high-level behavioral semantics in an interpretable and manipulatable way. We present a hierarchical framework that can effectively learn such sequential generative models. Our approach is inspired by recent work on leveraging programmatically produced weak labels, which we extend to the spatiotemporal regime. In addition to synthetic settings, we show how to instantiate our framework to effectively model complex interactions between basketball players and generate realistic multi-agent trajectories of basketball gameplay over long time periods. We validate our approach using both quantitative and qualitative evaluations, including a user study comparison conducted with professional sports analysts.

2.1 Introduction

The ongoing explosion of recorded tracking data is enabling the study of fine-grained behavior in many domains: sports (Miller et al., 2014; Yue et al., 2014; Zheng, Yue, and Lucey, 2016; Le et al., 2017), video games (Ross, Gordon, and J. Andrew Bagnell, 2011), video & motion capture (Suwajanakorn, Seitz, and Kemelmacher-Shlizerman, 2017; Taylor et al., 2017; Xue et al., 2016), navigation & driving (Ziebart et al., 2009; Zhang and Cho, 2017; Li, Song, and Ermon, 2017), laboratory animal behaviors (Johnson et al., 2016; Eyjolfsdottir et al., 2017), and tele-operated robotics (Abbeel and Ng, 2004; Lin et al., 2006). However, it is an open challenge to develop *sequential generative models* leveraging such data, for

instance, to capture the complex behavior of multiple cooperating agents. Figure 2.1a shows an example of offensive players in basketball moving unpredictably and with multimodal distributions over possible trajectories. Figure 2.1b depicts a simplified Boids model from (Reynolds, 1987) for modeling animal schooling behavior in which the agents can be friendly or unfriendly. In both cases, agent behavior is *highly coordinated and non-deterministic*, and the space of all multi-agent trajectories is naively exponentially large.

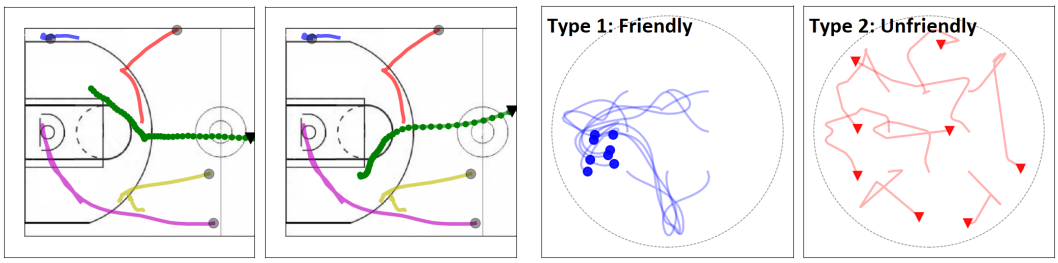
Conventional approaches to learning interpretable intermediate variables typically focus on learning disentangled latent representations in an unsupervised way (e.g., (Li, Song, and Ermon, 2017; Wang et al., 2017)), but it is challenging for such approaches to handle complex sequential settings (X. Chen et al., 2017).

To address this challenge, we present a hierarchical framework that can effectively learn such sequential generative models, while using programmatic weak supervision. Our approach uses a expert-written program to programmatically produce useful weak labels for supervised learning of interpretable intermediate representations. This approach is inspired by recent work on data programming (Ratner, Sa, et al., 2016), which uses cheap and noisy programs, which they call labeling functions, to significantly speed up learning. In this work, we extend this approach to the spatiotemporal regime.

Our contributions can be summarized as follows:

- We propose a hierarchical framework for sequential generative modeling. Our approach is compatible with many existing deep generative models.
- We show how to programmatically produce weak labels of macro-intents to train the intermediate representation in a supervised fashion. Our approach is easy to implement and results in highly interpretable intermediate variables, which allows for conditional inference by grounding macro-intents to manipulate behaviors.
- Focusing on multi-agent tracking data, we show that our approach can generate high-quality trajectories and effectively encode long-term coordination between multiple agents.

In addition to synthetic settings, we showcase our approach in an application on modeling team offense in basketball. We validate our approach both quantitatively and



(a) Offensive basketball players have multimodal behavior (ball not shown). For instance, the green player (▼) moves to either the top-left or bottom-left.

(b) Two types of generated behaviors for 8 agents in Boids model. **Left**: Friendly blue agents group together. **Right**: Unfriendly red agents stay apart.

Figure 2.1: Examples of coordinated multimodal multi-agent behavior in: a) professional basketball, and b) schooling behavior via the Boids model.

qualitatively, including a user study comparison with professional sports analysts, and show significant improvements over standard baselines.

2.2 Sequential Generative Modeling for Imitation Learning

In this work, we focus in motion tracking domains that have linear dynamics and trajectories as sequences of states. Let $\mathbf{s}_t \in \mathbb{R}^d$ denote the state at time t and $\tau = \{\mathbf{s}_1, \ldots, \mathbf{s}_T\}$ denote a trajectory of length T. Suppose we have a collection of N demonstrations: $\mathcal{D} = \{\tau_i\}_{i=1}^N$. In our experiments, all trajectories have the same length T, but in general this does not need to be the case.

The goal of sequential generative modeling for imitation learning is to learn a policy π_θ, with learnable parameters θ, that captures the distribution of trajectories in $\mathcal{D}$. A common approach is to factorize the joint distribution and then maximize the log-likelihood:

$$\theta^* = \arg\max_\theta \sum_{\tau \in \mathcal{D}} \log p(\tau) = \arg\max_\theta \sum_{\tau \in \mathcal{D}} \sum_{t=1}^{T} \log \pi_\theta(\mathbf{s}_t | \mathbf{s}_{<t}). \tag{2.1}$$

Here, $\mathbf{s}_{<t}$ refers to the subsequence of states in τ up to time t: $\mathbf{s}_{<t} = \{\mathbf{s}_1, \ldots, \mathbf{s}_{t-1}\}$. A common model choice for π_θ is a recurrent neural network.

Recurrent neural networks (RNN). An RNN models the conditional probabilities in Eq. (2.1) with a hidden state $\mathbf{h}_t$ that summarizes the information in the first $t-1$ timesteps:

$$\pi_\theta(\mathbf{s}_t | \mathbf{s}_{<t}) = \varphi(\mathbf{h}_{t-1}), \qquad \mathbf{h}_t = f(\mathbf{s}_t, \mathbf{h}_{t-1}), \tag{2.2}$$

where φ maps the hidden state to a probability distribution over states and f is a deterministic function such as LSTMs (Hochreiter and Schmidhuber, 1997) or GRUs (Cho et al., 2014).

Stochastic latent variable models. However, RNNs with simple output distributions that optimize Eq. (2.1) often struggle to capture highly variable and structured sequential data. For example, an RNN with Gaussian output distribution has difficulty learning the multimodal behavior of the green player moving to the top-left/bottom-left in Figure 2.1a. Recent work in sequential generative models address this issue by injecting stochastic latent variables into the model and optimizing using amortized variational inference to learn the latent variables (Fraccaro et al., 2016; Goyal et al., 2017; Chung et al., 2015). These models all stem from variational autoencoders (Kingma and Welling, 2014).

Variational autoencoders (VAE). A VAE (Kingma and Welling, 2014) is a generative model for data $\mathbf{x}$ that injects latent variables $\mathbf{z}$ into the joint distribution $p_\theta(\mathbf{x}, \mathbf{z})$ and introduces an inference network parametrized by ϕ to approximate the posterior $q_\phi(\mathbf{z} \mid \mathbf{x})$. The learning objective is to maximize the evidence lower-bound (ELBO) of the log-likelihood with respect to the model parameters θ and ϕ:

$$\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}|\mathbf{z})\right] - D_{KL}(q_\phi(\mathbf{z} \mid \mathbf{x})||p(\mathbf{z})). \tag{2.3}$$

The first term is known as the reconstruction term and can be approximated with Monte Carlo sampling. The second term is the Kullback-Leibler divergence between the approximate posterior and the prior, and can be evaluated analytically (i.e. if both distributions are Gaussian with diagonal covariance). The inference and generative models, $q_\phi(\mathbf{z} \mid \mathbf{x})$ and $p_\theta(\mathbf{x} \mid \mathbf{z})$ respectively, are often implemented with neural networks. In our imitation learning setting, input data $\mathbf{x}$ would be a trajectory and the generative model would be our policy.

Variational RNNs (VRNN). We use a VRNN (Chung et al., 2015) as our base model, but we emphasize that our approach is compatible with other sequential generative models as well. A VRNN is essentially a VAE conditioned on the hidden

state of an RNN (see Figure 2.3a):

$$p_\psi(\mathbf{z}_t|\mathbf{s}_{<t}, \mathbf{z}_{<t}) = \varphi_{\text{prior}}(\mathbf{h}_{t-1}) \quad \text{(prior)} \tag{2.4}$$

$$q_\phi(\mathbf{z}_t|\tau, \mathbf{z}_{<t}) = \varphi_{\text{enc}}(\mathbf{s}_t, \mathbf{h}_{t-1}) \quad \text{(inference)} \tag{2.5}$$

$$\pi_\theta(\mathbf{s}_t|\mathbf{z}_{\leq t}, \mathbf{s}_{<t}) = \varphi_{\text{dec}}(\mathbf{z}_t, \mathbf{h}_{t-1}) \quad \text{(generation/policy)} \tag{2.6}$$

$$\mathbf{h}_t = f(\mathbf{s}_t, \mathbf{z}_t, \mathbf{h}_{t-1}). \quad \text{(recurrence)} \tag{2.7}$$

VRNNs are also trained by maximizing the (sequential) ELBO, which in this case can be interpreted as the VAE ELBO summed over each timestep t:

$$\mathbb{E}_{q_\phi(\mathbf{z}_{\leq T}|\tau)}\left[\sum_{t=1}^{T} \log \pi_\theta(\mathbf{s}_t \mid \mathbf{z}_{\leq T}, \mathbf{s}_{<t}) - D_{KL}\Big(q_\phi(\mathbf{z}_t \mid \tau, \mathbf{z}_{<t})||p_\psi(\mathbf{z}_t \mid \mathbf{s}_{<t}, \mathbf{z}_{<t})\Big)\right]. \tag{2.8}$$

Note that the prior distribution of latent variable $\mathbf{z}_t$ depends on the history of states and latent variables (Eq. (2.4)). This temporal dependency of the prior allows VRNNs to model complex sequential data like speech and handwriting (Chung et al., 2015).

2.3 Hierarchical Framework using Macro-intents

In our multi-agent setting, we additionally assume that each trajectory τ consists of the trajectories of K coordinating agents. For example, the trajectory in Figure 2.1a can be decomposed into the trajectories of $K = 5$ basketball players. We denote $\tau = \{\tau^1, \ldots, \tau^K\}$, $\tau^k = \{\mathbf{s}_1^k, \ldots, \mathbf{s}_T^k\}$, and also $\mathbf{s}_t = \{\mathbf{s}_t^1, \ldots, \mathbf{s}_t^k\}$. Assuming conditional independence between the agent states $\mathbf{s}_t^k$ given state history $\mathbf{s}_{<t}$, we can factorize the maximum log-likelihood objective in Eq. (2.1) even further:

$$\theta^* = \arg\max_\theta \sum_{\tau \in \mathcal{D}} \sum_{t=1}^{T} \sum_{k=1}^{K} \log \pi_{\theta_k}(\mathbf{s}_t^k|\mathbf{s}_{<t}). \tag{2.9}$$

Naturally, there are two baseline approaches in this setting:

1. Treat τ as a single-agent trajectory and train a single model: $\theta = \theta_1 = \cdots = \theta_K$.
2. Train independent models for each agent: $\theta = \{\theta_1, \ldots, \theta_K\}$.

As we empirically verify in Section 2.4, VRNN models using these two approaches have difficulty learning representations of the data that generalize well over long time horizons, and capturing the coordination inherent in multi-agent trajectories. Our

solution introduces a hierarchical structure of *macro-intents* obtained via expert-written programs to effectively learn low-dimensional (distributional) representations of the data that extend in both time and space for multiple coordinating agents.

Defining macro-intents. We assume there exists shared latent variables called macro-intents that: 1) provide a tractable way to capture coordination between agents, 2) encode long-term intents of agents and enable long-term planning at a higher-level timescale, and 3) compactly represent some low-dimensional structure in an exponentially large multi-agent state space.

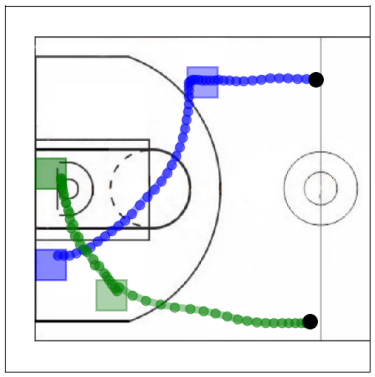


Figure 2.2: Macro-intents (boxes) for two basketball players. These macro-intents represent areas on the court that players move towards and can change over time.

For example, Figure 2.2 illustrates macro-intents for two basketball players as specific areas on the court (boxes). Upon reaching its macro-intent in the top-right, the blue player moves towards its next macro-intent in the bottom-left. Similarly, the green player moves towards its macro-intents from bottom-right to middle-left. These macro-intents are visible to both players and capture the coordination as they describe how the players plan to position themselves on the court. Macro-intents provide a compact summary of the players' trajectories over a long time.

Macro-intents do not need to have a geometric interpretation. For example, macro-intents in the Boids model in Figure 2.1b can be a binary label indicating friendly vs. unfriendly behavior. The goal is for macro-intents to encode long-term intent and ensure that agents behave more cohesively. Our modeling assumptions for macro-intents are:

- agent states $\{\mathbf{s}_t^k\}$ in an episode $[t_1, t_2]$ are conditioned on some shared macro-intent $\mathbf{g}_t$,
- the start and end times $[t_1, t_2]$ of episodes can vary between trajectories,

- macro-intents change slowly over time relative to the agent states: $d\mathbf{g}_t/dt \ll 1$,
- and due to their reduced dimensionality, we can model (near-)arbitrary dependencies between macro-intents (e.g., coordination) via black box learning.

Programs for macro-intent labels. Obtaining macro-intent labels from experts for training is ideal, but often too expensive. Instead, our work is inspired by recent advances in weak supervision settings known as *data programming*, in which multiple programs written by domain experts can be leveraged as weak and noisy label sources to learn the underlying structure of large unlabeled datasets (Ratner, Bach, et al., 2018; Bach et al., 2017). These programs often compute heuristics that allow users to incorporate domain knowledge into the model. For instance, the programs we use to obtain macro-intents for basketball trajectories compute the regions on the court in which players remain stationary; this integrates the idea that players aim to set up specific formations on the court. In general, programs can parse and label data very quickly, hence the name *programmatic weak supervision* in (Zhan et al., 2019).

Other approaches that try to learn macro-intents in a fully unsupervised learning setting can encounter difficulties that have been previously noted, such as the importance of choosing the correct prior and approximate posterior (Rezende and Mohamed, 2015) and the interpretability of learned latent variables (X. Chen et al., 2017). We find our approach using programs to be much more attractive, as it outperforms other baselines by generating samples of higher quality, while also avoiding the engineering required to address the aforementioned difficulties.

Hierarchical model with macro-intents Our hierarchical model uses an intermediate layer to model macro-intents, so our agent VRNN policies become:

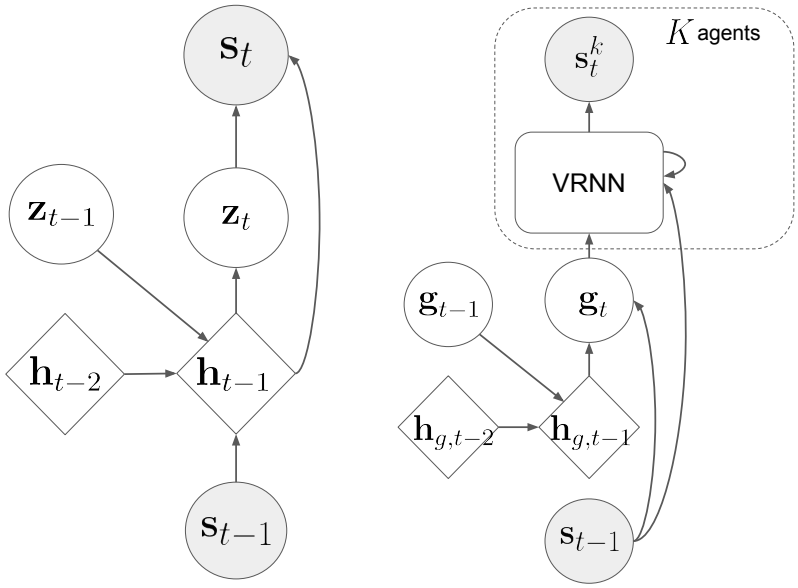
$$\pi_{\theta_k}(\mathbf{s}_t^k|\mathbf{s}_{<t}) = \varphi^k(\mathbf{z}_t^k, \mathbf{h}_{t-1}^k, \mathbf{g}_t), \tag{2.10}$$

where φ^k maps to a distribution over states, $\mathbf{z}_t^k$ is the VRNN latent variable, $\mathbf{h}_t^k$ is the hidden state of an RNN that summarizes the trajectory up to time t, and $\mathbf{g}_t$ is the shared macro-intent at time t. Figure 2.3b shows our hierarchical model, which samples macro-intents during generation rather than using only ground-truth macro-intents. Here, we train an RNN-model to sample macro-intents:

$$p(\mathbf{g}_t|\mathbf{g}_{<t}) = \varphi_g(\mathbf{h}_{g,t-1}, \mathbf{s}_{t-1}), \tag{2.11}$$

where φ_g maps to a distribution over macro-intents and $\mathbf{h}_{g,t-1}$ summarizes the history of macro-intents up to time t. We condition the macro-intent model on previous states $\mathbf{s}_{t-1}$ in Eq. (2.11) and generate next states by first sampling a macro-intent $\mathbf{g}_t$, and then sampling $\mathbf{s}_t^k$ conditioned on $\mathbf{g}_t$ (see Figure 2.3b). Note that all agent-models for generating $\mathbf{s}_t^k$ share the same macro-intent variable $\mathbf{g}_t$. This is core to our approach as it induces coordination between agent trajectories (see Section 2.4).

We learn our agent-models by maximizing the VRNN objective from Eq (2.8) conditioned on the shared $\mathbf{g}_t$ variables while independently learning the macro-intent model via supervised learning by maximizing the log-likelihood of macro-intent labels obtained programmatically.



(a) VRNN (b) Our model with macro-intents $\mathbf{g}_t$

Figure 2.3: Depicting VRNN and our model. Circles are stochastic and diamonds are deterministic. Macro-intent $\mathbf{g}_t$ is shared across agents. In principle, any generative model can be used in our framework.

2.4 Experiments

We first apply our approach on generating offensive team basketball gameplay (team with possession of the ball), and then on a synthetic Boids model dataset. We present both quantitative and qualitative experimental results. Our quantitative results include a user study comparison with professional sports analysts, who significantly preferred basketball rollouts generated from our approach to standard baselines. Our qualitative results demonstrate the ability of our approach to generate high-quality rollouts under various conditions.

Experimental Setup for Basketball

Training data. Each demonstration in our data contains trajectories of $K = 5$ players on the left half-court, recorded for $T = 50$ timesteps at 6 Hz. The offensive team has possession of the ball for the entire sequence. $\mathbf{s}_t^k$ are the coordinates of player k at time t on the court (50×94 feet). We normalize and mean-shift the data. Players are ordered based on their relative positions, similar to the role assignment in (Lucey et al., 2013). There are 107,146 training and 13,845 test examples. We ignore the defensive players and the ball to focus on capturing the coordination and multimodality of the offensive team. In principle, we can provide the defensive positions as conditional input for our model and update the defensive positions using methods such as (Le et al., 2017). We leave the task of modeling the ball and defense for future work.

Macro-intent program. We extract weak macro-intent labels $\hat{\mathbf{g}}_t^k$ for each player k as done in (Zheng, Yue, and Lucey, 2016). We segment the left half-court into a 10×9 grid of 5ft $\times$ 5ft boxes. The weak macro-intent $\hat{\mathbf{g}}_t^k$ at time t is a one-hot encoding of dimension 90 of the next box in which player k is stationary (speed $\left\|\mathbf{s}_{t+1}^k - \mathbf{s}_t^k\right\|_2$ below a set threshold). The shared global macro-intent $\mathbf{g}_t$ is the concatenation of individual macro-intents. Figure 2.4 shows the distribution of macro-intents for each player. We refer to this macro-intent program as `Stationary` (pseudocode in Appendix A.3).

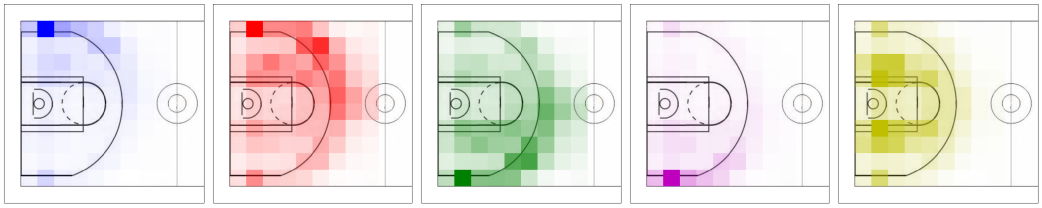


Figure 2.4: Distribution of weak macro-intent labels extracted for each player from the training data. Color intensity corresponds to frequency of macro-intent label. Players are ordered by their relative positions on the court, which can be seen from the macro-intent distributions.

Model details. We model each latent variable $\mathbf{z}_t^k$ as a multivariate Gaussian with diagonal covariance of dimension 16. All output models are implemented with memory-less 2-layer fully-connected neural networks with a hidden layer of size 200. Our agent-models sample from a multivariate Gaussian with diagonal covariance while our macro-intent models sample from a multinomial distribution over the macro-intents. All hidden states $(\mathbf{h}_{g,t}, \mathbf{h}_t^1, \ldots \mathbf{h}_t^K)$ are modeled with 200 2-layer

GRU memory cells each. We maximize the log-likelihood/ELBO with stochastic gradient descent using the Adam optimizer (Kingma and Ba, 2015) and a learning rate of 0.0001.

Baselines. We compare with 5 baselines that do not use macro-intents from programs:

1. **RNN-gauss:** RNN without latent variables using 900 2-layer GRU cells as the hidden state.
2. **VRNN-single:** VRNN in which we concatenate all player positions together ($K = 1$) with 900 2-layer GRU cells for the hidden state and a 80-dim latent variable.
3. **VRNN-indep:** VRNN for each agent with 250 2-layer GRUs and 16-dim latent variables.
4. **VRNN-mixed:** Combination of VRNN-single and VRNN-indep. The shared hidden state of 600 2-layer GRU cells is fed into decoders with 16-dim latent variables for each agent.
5. **VRAE-mi:** VRAE-style architecture (Fabius and Amersfoort, 2014) that maximizes the mutual information between τ and a single global macro-intent. We refer to Appendix A.2 for details.

Quantitative Evaluation for Basketball

Log-likelihood. Table 2.1 reports the average log-likelihoods on the test data. Our approach outperforms RNN-gauss and is comparable with other baselines. However, higher log-likelihoods do not necessarily indicate higher quality of generated samples (Theis, van den Oord, and Bethge, 2015). As such, we also evaluate using other means, such as human preference studies and auxiliary statistics.

Human preference study. We recruited 14 professional sports analysts as judges to compare the quality of policy rollouts. Each comparison animates two rollouts, one from our model and another from a baseline. Both rollouts are burned-in for 10 timesteps with the same ground-truth states from the test set, and then generated for the next 40 timesteps. Judges decide which of the two rollouts looks more realistic. Table 2.2 shows the results from the preference study. We tested our model against two baselines, VRNN-single and VRNN-indep, with 25 comparisons for each. All judges preferred our model over the baselines with 98% statistical

Model	Basketball	Boids
RNN-gauss	1931	2414
VRNN-single	$\geq$ 2302	$\geq$ 2417
VRNN-indep	$\geq$ 2360	$\geq$ 2385
VRNN-mixed	$\geq$ 2323	$\geq$ 2204
VRAE-mi	$\geq$ 2349	$\geq$ 2331
Ours	$\geq$ **2362**	$\geq$ **2428**

Table 2.1: Average log-likelihoods per test sequence. "$\geq$" indicates ELBO of log-likelihood. Our hierarchical model achieves higher log-likelihoods than baselines for both datasets.

significance. These results suggest that our model generates rollouts of significantly higher quality than the baselines.

vs. Model	Win/Tie/Loss	Avg Gain
vs. VRNN-single	25/0/0	0.57
vs. VRNN-indep	15/4/6	0.23

Table 2.2: Basketball preference study results. Win/Tie/Loss indicates how often our model is preferred over baselines (25 comparisons per baseline). Gain is computed by scoring +1 when our model is preferred and -1 otherwise. Results are 98% significant using a one-sample t-test.

Domain statistics. Finally, we compute several basketball statistics (average speed, average total distance traveled, % of frames with players out-of-bounds) and summarize them in Table 2.3. Our model generates trajectories that are most similar to ground-truth trajectories with respect to these statistics, indicating that our model generates significantly more realistic behavior than all baselines.

Choice of program for macro-intents. In addition to `Stationary`, we also assess the quality of our approach using macro-intents obtained from different programs. `Window25` and `Window50` labels macro-intents as the last region a player resides in every window of 25 and 50 timesteps respectively (pseudocode in Appendix A.3). Table 2.3 shows that domain statistics from our models using programmatic weak supervision match closer to the ground-truth with more informative labeling functions (`Stationary` > `Window25` > `Window50`). This is expected, since `Stationary` provides the most information about the structure of the data.

Qualitative Evaluation of Generated Policy Rollouts for Basketball

We next conduct a qualitative visual inspection of policy rollouts. Figures 2.5 and 2.6 show rollouts generated from VRNN-single, VRNN-indep, and our model by

Model	Speed (ft)	Distance (ft)	Out-of-bounds (%)
RNN-gauss	3.05	149.57	46.93
VRNN-single	1.28	62.67	45.67
VRNN-indep	0.89	43.78	33.78
VRNN-mixed	0.91	44.80	27.19
VRAE-mi	0.98	48.25	20.09
Ours (`Window50`)	0.99	48.53	28.84
Ours (`Window25`)	0.87	42.99	**14.53**
Ours (`Stationary`)	**0.79**	**38.92**	15.52
Ground-truth	0.77	37.78	2.21

Table 2.3: Domain statistics of 1000 basketball trajectories generated from each model: average speed, average distance traveled, and % of frames with players out-of-bounds. Trajectories from our models using programmatic weak supervision match the closest with the ground-truth.

sampling states for 40 timesteps after an initial burn-in period of 10 timesteps with ground-truth states from the test set.

Common problems in baseline rollouts include players moving out of bounds or in the wrong direction (Figure 2.5). These issues tend to occur at later timesteps, suggesting that the baselines do not perform well over long horizons. One possible explanation is due to compounding errors (Ross, Gordon, and J. Andrew Bagnell, 2011): if the model makes a mistake and deviates from the states seen during training, it is likely to make more mistakes in the future and generalize poorly. On the other hand, generated rollouts from our model are more robust to the types of errors made by the baselines (Figure 2.6).

Macro-intents induce multimodal and interpretable rollouts. Generated macro-intents allow us to interpret the intent of each individual player as well as a global team strategy (e.g. setting up a specific formation on the court). We highlight that our model learns a multimodal generating distribution, as repeated rollouts with the same burn-in result in a dynamic range of generated trajectories, as seen in Figure 2.7 Left. Furthermore, Figure 2.7 Right demonstrates that grounding macro-intents during generation instead of sampling them allows us to control agent behavior.

Macro-intents induce coordination. Figure 2.8 illustrates how the macro-intents encode coordination between players that results in realistic rollouts of players moving cohesively. As we change the trajectory and macro-intent of the red player, the distribution of macro-intents generated from our model for the green player changes such that the two players occupy different areas of the court.

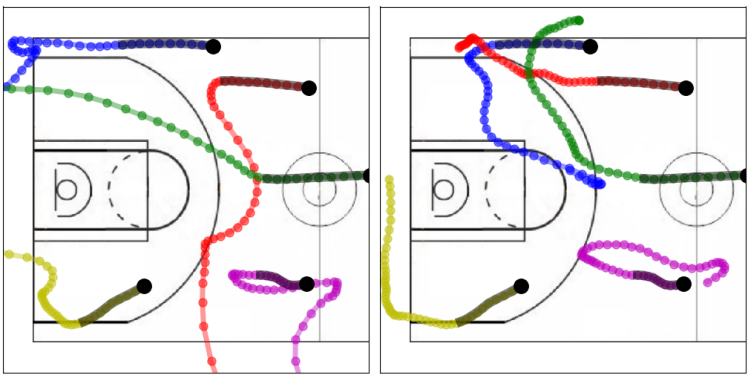


Figure 2.5: Baseline rollouts of representative quality starting from black dots, generated for 40 timesteps after an initial burn-in period of 10 timesteps (marked by dark shading). **Left**: VRNN-single. **Right**: VRNN-indep. Common problems in baseline rollouts include players moving out of bounds or in the wrong direction. Players do not appear to behave cohesively as a team.

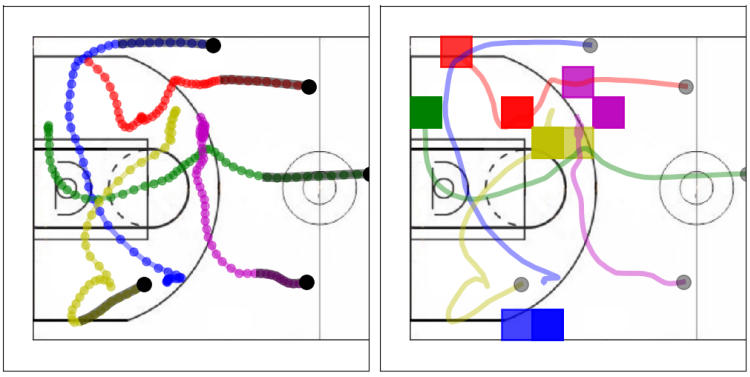


Figure 2.6: **Left**: Rollout from our model starting from black dots, generated for 40 timesteps after an initial burn-in period of 10 timesteps (marked by dark shading). All players remain in bounds. **Right**: Corresponding macro-intents for left rollout. Macro-intent generation is stable and suggests that the team is creating more space for the blue player (perhaps setting up an isolation play).

Synthetic Experiments: Boids Model of Schooling Behavior

To illustrate the generality of our approach, we apply our model to a simplified version of the Boids model (Reynolds, 1987) that produces realistic trajectories of schooling behavior. We generate trajectories for 8 agents for 50 frames. The agents start in fixed positions around the origin with initial velocities sampled from a unit Gaussian. Each agent's velocity is then updated at each timestep:

$$\mathbf{v}_{t+1} = \beta \mathbf{v}_t + \beta(c_1 \mathbf{v}_{\text{coh}} + c_2 \mathbf{v}_{\text{sep}} + c_3 \mathbf{v}_{\text{ali}} + c_4 \mathbf{v}_{\text{ori}}). \tag{2.12}$$

Full details of the model can be found in Appendix A.1. We randomly sample the sign of c_1 for each trajectory, which produces two distinct types of behaviors:

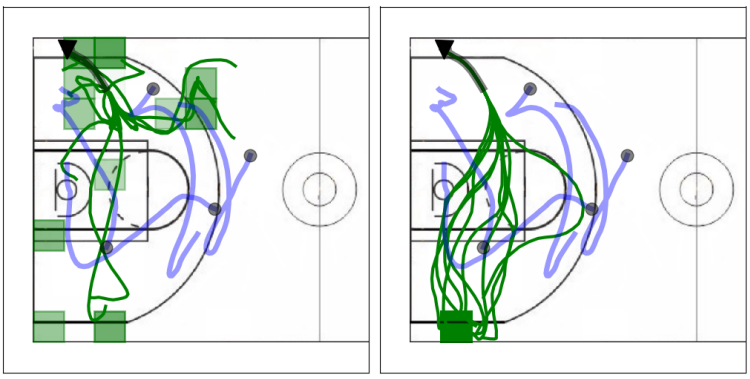


Figure 2.7: 10 rollouts of the green player (▼) with a burn-in period of 20 timesteps. Blue trajectories are fixed and (•) indicates initial positions. **Left**: The model generates macro-intents. **Right**: We ground the macro-intents at the bottom-left. In both, we observe a multimodal distribution of trajectories.

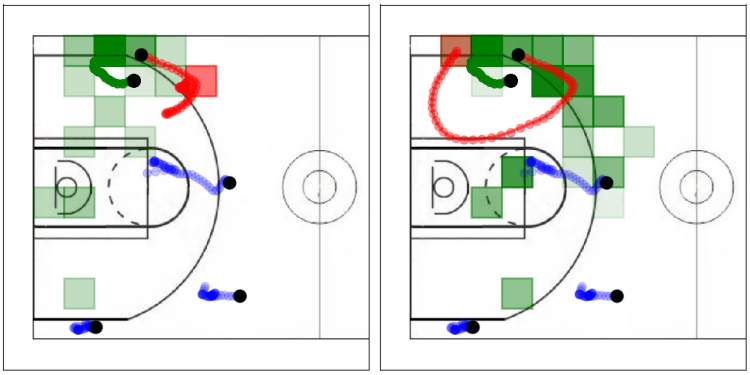


Figure 2.8: The distribution of macro-intents sampled from 20 rollouts of the green player changes in response to the change in red trajectories and macro-intents. This suggests that macro-intents encode and induce coordination between multiple players. Blue trajectories are fixed and (•) indicates initial positions.

friendly agents ($c_1 > 0$) that like to group together, and *unfriendly agents* ($c_1 < 0$) that like to stay apart (see Figure 2.1b). We also introduce more stochasticity into the model by periodically updating β randomly.

Our macro-intent program thresholds the average distance to an agent's closest neighbor (see last plot in Figure 2.9). This is equivalent to using the sign of c_1 as our macro-intents, which indicates the type of behavior. Note that unlike our macro-intents for the basketball dataset, these macro-intents are simpler and have no geometric interpretation. All models have similar average log-likelihoods on the test set in Table 2.1, but our hierarchical model can capture the true generating distribution much better than the baselines. For example, Figure 2.9 depicts the histograms of average distances to an agent's closest neighbor in trajectories generated

from all models and the ground-truth. Our model more closely captures the two distinct modes in the ground-truth (friendly, small distances, left peak vs. unfriendly, large distances, right peak) whereas the baselines fail to distinguish them.

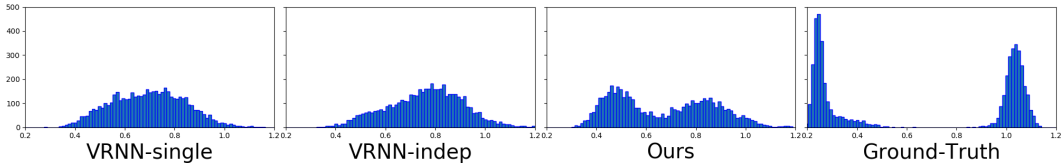


Figure 2.9: Synthetic Boids experiments. Showing histograms (horizontal axis: distance; vertical: counts) of average distance to an agent's closest neighbor in 5000 rollouts. Our hierarchical model more closely captures the two distinct modes for friendly (small distances, left peak) vs. unfriendly (large distances, right peak) behavior compared to baselines, which do not learn to distinguish them.

Inspecting the Hierarchical Model Class

Output distribution for states. The outputs of all models (including baselines) sample from a multivariate Gaussian with diagonal covariance. We also experimented with sampling from a mixture of 2, 3, 4, and 8 Gaussian components, but discovered that the models would always learn to assign all the weight on a single component and ignore the others. The variance of the active component is also very small. This is intuitive because sampling with a large variance at every timestep would result in noisy trajectories and not the smooth ones that we see in Figures 2.6, 2.7.

Choice of macro-intent model. In principle, we can use more expressive generative models, like a VRNN, to model macro-intents over richer macro-intent spaces in Eq. (2.11). In our case, we found that an RNN was sufficient in capturing the distribution of macro-intents shown in Figure 2.4. The RNN learns multinomial distributions over macro-intents that are peaked at a single macro-intent and relatively static through time, which is consistent with the macro-intent labels that we extracted from data. Latent variables in a VRNN had minimal effect on the multinomial distribution.

Maximizing mutual information isn't effective. The learned macro-intents in our fully unsupervised VRAE-mi model do not encode anything useful and are essentially ignored by the model. In particular, the model learns to match the approximate posterior of macro-intents from the encoder with the discriminator from the mutual information lower-bound. This results in a lack of diversity in

rollouts as we vary the macro-intents during generation. We refer to Appendix A.2 for examples.

2.5 Related Work

Deep generative models. The study of deep generative models is an increasingly popular research area, due to their ability to inherit both the flexibility of deep learning and the probabilistic semantics of generative models. In general, there are two ways that one can incorporate stochastics into deep models. The first approach models an explicit distribution over actions in the output layer, e.g., via logistic regression (L.-C. Chen et al., 2015; Oord, Dieleman, et al., 2016; Oord, Kalchbrenner, and Kavukcuoglu, 2016; Zheng, Yue, and Lucey, 2016; Eyjolfsdottir et al., 2017). The second approach uses deep neural nets to define a transformation from a simple distribution to one of interest (Goodfellow et al., 2014; Kingma and Welling, 2014; Rezende, Mohamed, and Wierstra, 2014) and can more readily be extended to incorporate additional structure, such as a hierarchy of random variables (Ranganath, Tran, and Blei, 2016) or dynamics (Johnson et al., 2016; Chung et al., 2015; Krishnan, Shalit, and Sontag, 2017; Fraccaro et al., 2016). Our framework can incorporate both variants.

Structured probabilistic models. Recently, there has been increasing interest in probabilistic modeling with additional structure or side information. Existing work includes approaches that enforce logic constraints (Akkaya et al., 2016), specify generative models as programs (Tran et al., 2016), or automatically produce weak supervision via data programming (Ratner, Sa, et al., 2016). Our framework is inspired by the latter, which we extend to the spatiotemporal regime. Additionally, probabilistic models with multiresolution hierarchical structure has also been studied in the context of missing value imputation for spatiotemporal domains (Liu et al., 2019).

Imitation learning. Our work is also related to imitation learning, which aims to learn a policy that can mimic demonstrated behavior (Syed and Schapire, 2008; Abbeel and Ng, 2004; Ziebart et al., 2008; Ho and Ermon, 2016). There has been some prior work in multi-agent imitation learning (Le et al., 2017; Song et al., 2018) and learning stochastic policies (Ho and Ermon, 2016; Li, Song, and Ermon, 2017), but no previous work has focused on learning generative polices while simultaneously addressing generative and multi-agent imitation learning. For instance, experiments in (Ho and Ermon, 2016) all lead to highly peaked distributions, while

(Li, Song, and Ermon, 2017) captures multimodal distributions by learning unimodal policies for a fixed number of experts. (Hrolenok, Boots, and Balch, 2017) raise the issue of learning stochastic multi-agent behavior, but their solution involves significant feature engineering.

2.6 Discussion

The programs for labeling macro-intents used in our experiments are relatively simple. For instance, rather than simply using location-based macro-intents, we can also incorporate complex interactions such as "pick and roll". Another future direction is to explore how to adapt our method to different domains, e.g., defining a macro-intent representing "argument" for a dialogue between two agents, or a macro-intent representing "refrain" for music generation for "coordinating instruments" (Thickstun, Harchaoui, and Kakade, 2017). We have shown that weak macro-intent labels extracted using simple domain-specific heuristics can be effectively used to generate high-quality coordinated multi-agent trajectories. An interesting direction is to incorporate multiple programs simultaneously, each viewed as noisy realizations of true macro-intents, similar to (Ratner, Sa, et al., 2016; Ratner, Bach, et al., 2018; Bach et al., 2017).

References

Abbeel, Pieter and Andrew Y Ng (2004). "Apprenticeship learning via inverse reinforcement learning". In: *Proceedings of the twenty-first international conference on Machine learning*, p. 1.

Akkaya, Ilge et al. (2016). "Control improvisation with probabilistic temporal specifications". In: *2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI)*.

Bach, Stephen H. et al. (2017). "Learning the Structure of Generative Models without Labeled Data". In: *ICML*.

Chen, Liang-Chieh et al. (2015). "Learning deep structured models". In: *ICML*.

Chen, Xi et al. (2017). "Variational Lossy Autoencoder". In: *ICLR*.

Cho, Kyunghyun et al. (2014). "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *arXiv preprint arXiv:1406.1078*.

Chung, Junyoung et al. (2015). "A Recurrent Latent Variable Model for Sequential Data". In: *NIPS*.

Eyjolfsdottir, Eyrun et al. (2017). "Learning recurrent representations for hierarchical behavior modeling". In: *International Conference on Learning Representations*.

Fabius, Otto and Joost R van Amersfoort (2014). "Variational recurrent auto-encoders". In: *ICLR workshop*.

Fraccaro, Marco et al. (2016). "Sequential Neural Models with Stochastic Layers". In: *NIPS*.

Goodfellow, Ian et al. (2014). "Generative adversarial nets". In: *NIPS*.

Goyal, Anirudh et al. (2017). "Z-Forcing: Training Stochastic Recurrent Networks". In: *NIPS*.

Ho, Jonathan and Stefano Ermon (2016). "Generative Adversarial Imitation Learning". In: *NIPS*.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural Computation* 9.8, pp. 1735–1780.

Hrolenok, Brian, Byron Boots, and Tucker Balch (2017). "Sampling Beats Fixed Estimate Predictors for Cloning Stochastic Behavior in Multiagent Systems". In: *AAAI*.

Johnson, Matthew J et al. (2016). "Composing graphical models with neural networks for structured representations and fast inference". In: *Advances in Neural Information Processing Systems*.

Kingma, Diederik P. and Jimmy Ba (2015). "Adam: A Method for Stochastic Optimization". In: *ICLR*.

Kingma, Diederik P. and Max Welling (2014). "Auto-Encoding Variational Bayes". In: *ICLR*.

Krishnan, Rahul G., Uri Shalit, and David Sontag (2017). "Structured Inference Networks for Nonlinear State Space Models". In: *AAAI*.

Le, Hoang Minh et al. (2017). "Coordinated Multi-Agent Imitation Learning". In: *ICML*.

Li, Yunzhu, Jiaming Song, and Stefano Ermon (2017). "Infogail: Interpretable imitation learning from visual demonstrations". In: *NIPS*.

Lin, Henry C et al. (2006). "Towards automatic skill evaluation: Detection and segmentation of robot-assisted surgical motions". In: *Computer Aided Surgery* 11.5, pp. 220–230.

Liu, Yukai et al. (2019). "Naomi: Non-autoregressive multiresolution sequence imputations". In: *Advances in Neural Information Processing Systems*. Vol. 32. URL: https://papers.nips.cc/paper/2019/hash/50c1f44e426560f3f2cdcb3e19e39903-Abstract.html.

Lucey, Patrick et al. (2013). "Representing and discovering adversarial team behaviors using player roles". In: *CVPR*.

Miller, Andrew et al. (2014). "Factorized point process intensities: A spatial analysis of professional basketball". In: *ICML*.

Oord, Aaron van den, Sander Dieleman, et al. (2016). “Wavenet: A generative model for raw audio”. In: *arXiv preprint arXiv:1609.03499*.

Oord, Aaron van den, Nal Kalchbrenner, and Koray Kavukcuoglu (2016). “Pixel recurrent neural networks”. In: *ICML*.

Ranganath, Rajesh, Dustin Tran, and David Blei (2016). “Hierarchical variational models”. In: *ICML*.

Ratner, Alexander, Stephen H. Bach, et al. (2018). “Snorkel: Rapid Training Data Creation with Weak Supervision”. In: *VLDB*.

Ratner, Alexander, Christopher De Sa, et al. (2016). “Data Programming: Creating Large Training Sets, Quickly”. In: *NIPS*.

Reynolds, Craig W. (1987). “Flocks, Herds and Schools: A Distributed Behavioral Model”. In: *SIGGRAPH*.

Rezende, Danilo Jimenez and Shakir Mohamed (2015). “Variational Inference with Normalizing Flows”. In: *ICML*.

Rezende, Danilo Jimenez, Shakir Mohamed, and Daan Wierstra (2014). “Stochastic backpropagation and approximate inference in deep generative models”. In: *ICML*.

Ross, Stéphane, Geoffrey J. Gordon, and J. Andrew Bagnell (2011). “No-Regret Reductions for Imitation Learning and Structured Prediction”. In: *AISTATS*.

Song, Jiaming et al. (2018). “Multi-agent generative adversarial imitation learning”. In: *NIPS*.

Suwajanakorn, Supasorn, Steven M Seitz, and Ira Kemelmacher-Shlizerman (2017). “Synthesizing obama: learning lip sync from audio”. In: *ACM Transactions on Graphics (TOG)* 36.4, p. 95.

Syed, Umar and Robert E Schapire (2008). “A game-theoretic approach to apprenticeship learning”. In: *NIPS*.

Taylor, Sarah et al. (2017). “A deep learning approach for generalized speech animation”. In: *SIGGRAPH*.

Theis, L., A. van den Oord, and M. Bethge (2015). “A note on the evaluation of generative models”. In: *arXiv preprint arXiv:1511.01844*.

Thickstun, John, Zaid Harchaoui, and Sham Kakade (2017). “Learning Features of Music from Scratch”. In: *ICLR*.

Tran, Dustin et al. (2016). “Edward: A library for probabilistic modeling, inference, and criticism”. In: *arXiv preprint arXiv:1610.09787*.

Wang, Ziyu et al. (2017). “Robust Imitation of Diverse Behaviors”. In: *arXiv preprint arXiv:1707.02747*.

Xue, Tianfan et al. (2016). “Visual dynamics: Probabilistic future frame synthesis via cross convolutional networks”. In: *NIPS*.

Yue, Yisong et al. (2014). “Learning fine-grained spatial models for dynamic sports play prediction”. In: *ICDM*.

Zhan, Eric et al. (2019). “Generating Multi-Agent Trajectories using Programmatic Weak Supervision”. In: *International Conference on Learning Representation*. URL: https://openreview.net/forum?id=rkxw-hAcFQ.

Zhang, Jiakai and Kyunghyun Cho (2017). “Query-Efficient Imitation Learning for End-to-End Autonomous Driving.” In: *AAAI*.

Zheng, Stephan, Yisong Yue, and Patrick Lucey (2016). “Generating Long-term Trajectories Using Deep Hierarchical Networks”. In: *NIPS*.

Ziebart, Brian D et al. (2008). “Maximum entropy inverse reinforcement learning.” In: *Aaai*. Vol. 8, pp. 1433–1438.

Ziebart, Brian D et al. (2009). “Human Behavior Modeling with Maximum Entropy Inverse Optimal Control”. In: *AAAI*.

Chapter 3

LEARNING CONTROLLABLE STYLE-CONSISTENT POLICIES

The work in this chapter was published in (Zhan, Tseng, et al., 2020). E.Z. participated in the conception of the project, formulated and implemented the method, conducted experiments and analyzed results, and participated in the writing of the manuscript.

Summary

We study the problem of controllable generation of long-term sequential behaviors, where the goal is to calibrate to multiple behavior styles simultaneously. In contrast to the well-studied areas of controllable generation of images, text, and speech, there are two questions that pose significant challenges when generating long-term behaviors: how should we specify the factors of variation to control, and how can we ensure that the generated behavior faithfully demonstrates combinatorially many styles? We leverage expert-written programs to specify controllable styles, and derive a formal notion of style-consistency as a learning objective, which can then be solved using conventional policy learning approaches. We evaluate our framework using demonstrations from professional basketball players and agents in the MuJoCo physics environment, and show that existing approaches that do not explicitly enforce style-consistency fail to generate diverse behaviors whereas our learned policies can be calibrated for up to $4^5(1024)$ distinct style combinations.

3.1 Introduction

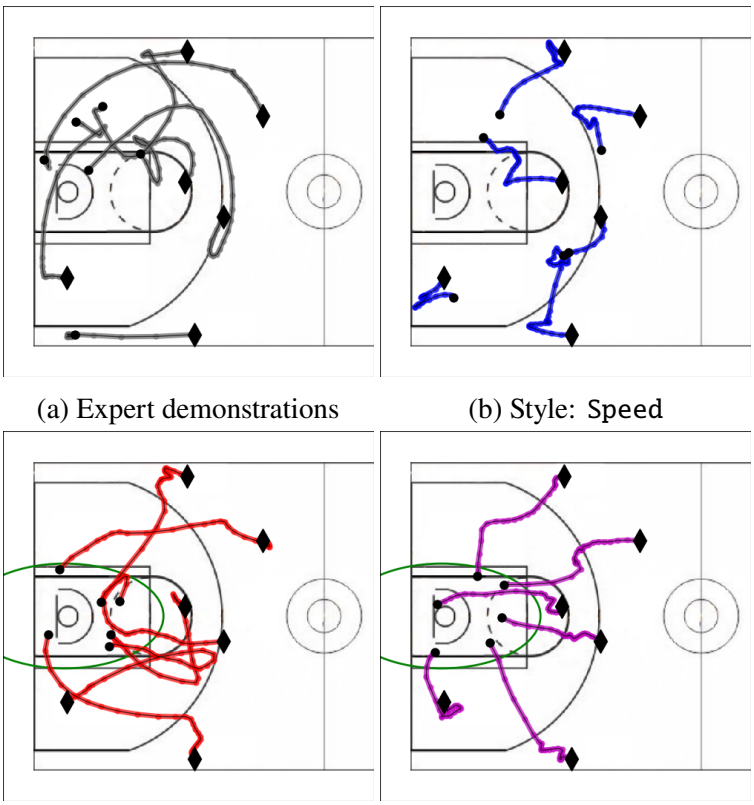
The widespread availability of recorded tracking data is enabling the study of complex behaviors in many domains, including sports (J. Chen et al., 2016; Le, Yue, et al., 2017; Zhan, Zheng, et al., 2019; Yeh et al., 2019), video games (Kurin et al., 2017; Broll et al., 2019; Hofmann, 2019), laboratory animals (Eyjolfsdottir, S. Branson, et al., 2014; Eyjolfsdottir, K. Branson, et al., 2017; K. Branson et al., 2009; Johnson et al., 2016), facial expressions (Suwajanakorn, Seitz, and Kemelmacher-Shlizerman, 2017; Taylor et al., 2017), commonplace activities such as cooking (Nishimura et al., 2019), and transportation (Bojarski et al., 2016; W. Luo, Yang, and Urtasun, 2018; Yaguang Li et al., 2018; Chang et al., 2019). A key aspect of modern behavioral datasets is that the behaviors can exhibit very diverse styles (e.g.,

from multiple demonstrators). For example, Figure 3.1a depicts demonstrations from basketball players with variations in speed, desired destinations, and curvature of movement.

The goal of this chapter is to study controllable generation of diverse behaviors by learning to imitate raw demonstrations; or more technically, to develop style-calibratable imitation learning methods. A controllable, or *calibratable*, policy would enable the generation of behaviors consistent with various styles, such as low movement speed (Figure 3.1b), or approaching the basket (Figure 3.1c), or both styles simultaneously (Figure 3.1d). Style-calibrated imitation learning methods that can yield such policies can be broadly useful to: (a) perform more robust imitation learning from diverse demonstrations (Z. Wang et al., 2017; Broll et al., 2019), (b) enable diverse exploration in reinforcement learning agents (Co-Reyes et al., 2018), or (c) visualize and extrapolate counterfactual behaviors beyond those seen in the dataset (Le, Carr, et al., 2017), amongst many other tasks.

Performing style-calibrated imitation is a challenging task. First, what constitutes a "style"? Second, when can we be certain that a policy is successfully calibrated for imitating a style? Third, how can we scale policy learning to faithfully generate combinatorially many styles? In related tasks like controllable image generation, common approaches for calibration use adversarial information factorization or mutual information between generated images and user-specified styles (e.g. gender, hair length, etc.) (Creswell, Bharath, and Sengupta, 2017; Lample et al., 2017; X. Chen et al., 2016). However, we find that these *indirect* approaches fall well short of generating controllable sequential behaviors. Intuitively, the aforementioned objectives provide only indirect proxies for style-calibration. For example, Figure 3.2 illustrates that an indirect baseline approach struggles to reliably generate trajectories to reach a certain displacement, even though the dataset contains many examples of such behavior.

Research questions. We seek to answer three research questions while tackling this challenge. The first is strategic: since high-level stylistic attributes like movement speed are typically not provided with the raw demonstration data, what systematic form of domain knowledge can we leverage to quickly and cleanly extract highly varied style information from raw behavioral data? The second is formulaic: how can we formalize the learning objective to encourage learning style-calibratable policies that can be controlled to realize many diverse styles? The third is algorithmic: how do we design practical learning approaches that reliably optimize the learning



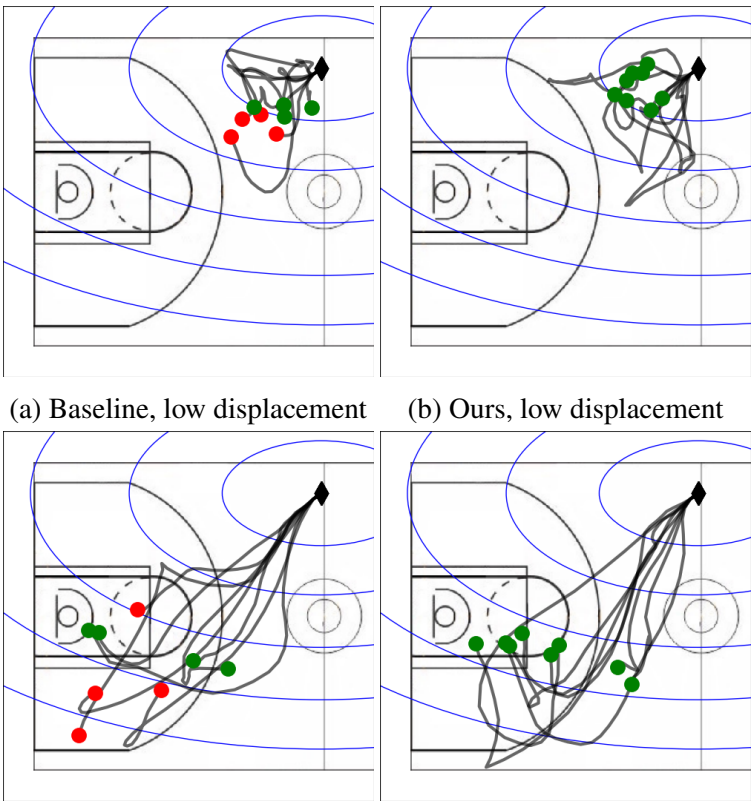
(a) Expert demonstrations (b) Style: `Speed`

(c) Style: `Destination` (d) Both styles

Figure 3.1: Basketball trajectories from policies that are (a) from the expert, (b) calibrated to move at low speeds, (c) calibrated to end near the basket (within green boundary), and (d) calibrated for both (b,c) simultaneously. Diamonds (♦) and dots (•) are initial and final positions.

objective?

Our contributions. To address these questions, we present a novel framework inspired by *data programming* (Ratner et al., 2016), a paradigm in weak supervision that utilizes automated labeling procedures, called labeling functions or programs, to learn without ground-truth labels. In our setting, programs enable domain experts to quickly translate domain knowledge of diverse styles into programmatically generated style annotations. For instance, it is trivial to write a program for the styles depicted in Figures 3.1 & 3.2 (speed and destination). Programs also motivate a new learning objective, which we call *programmatic style-consistency*: rollouts generated by a policy calibrated for a particular style should return the same style label when fed back into the program. This notion of style-consistency provides a *direct* approach to measuring how calibrated a policy is, and does not suffer from the weaknesses of indirect approaches such as mutual information estimation. In



(a) Baseline, low displacement (b) Ours, low displacement

(c) Baseline, high displacement (d) Ours, high displacement

Figure 3.2: Basketball trajectories sampled from baseline policies and our models calibrated to the style of `Displacement` with 6 classes corresponding to regions separated by blue lines. Diamonds (♦) and dots (•) indicate initial and final positions respectively. Each policy is conditioned on a label class for `Displacement` (low in (a,b), high in (c,d)). Green dots indicate trajectories that are consistent with the style label, while red dots indicate those that are not. Our policy (b,d) is better calibrated for this style than the baselines (a,c).

the basketball example of scoring when near the basket, trajectories that perform correlated events (like turning towards the basket) will not return the desired style label when fed to the program that checks for scoring events. We elaborate on this in Section 3.3.

We demonstrate style-calibrated policy learning in Basketball and MuJoCo domains. Our experiments highlight the modularity of our approach—we can plug in any policy class and any imitation learning algorithm and reliably optimize for style-consistency using the approach of Section 3.4. The resulting learned policies can achieve very fine-grained and diverse style-calibration with negligible degradation in imitation quality—for example, our learned policy is calibrated to $4^5(1024)$ distinct style combinations in Basketball.

3.2 Imitation Learning for Behavior Modeling

Since our focus is on learning style-calibratable generative policies, for simplicity we develop our approach with the basic imitation learning paradigm of behavioral cloning. Interesting future directions include composing our approach with more advanced imitation learning approaches like DAgger (Ross, Gordon, and Bagnell, 2011) and GAIL (Ho and Ermon, 2016), as well as with reinforcement learning.

Notation. Let $\mathcal{S}$ and $\mathcal{A}$ denote the environment state and action spaces. At each timestep t, an agent observes state $\mathbf{s}_t \in \mathcal{S}$ and executes action $\mathbf{a}_t \in \mathcal{A}$ using a policy $\pi : \mathcal{S} \to \mathcal{A}$. The environment then transitions to the next state $\mathbf{s}_{t+1}$ according to a (typically unknown) dynamics function $P : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. For the rest of this paper, we assume P is deterministic; a modification of our approach for stochastic P is included in Appendix B.2. We represent a trajectory τ as a sequence of T state-action pairs and the last state: $\tau = \{(\mathbf{s}_t, \mathbf{a}_t)\}_{t=1}^{T} \cup \{\mathbf{s}_{T+1}\}$. Let $\mathcal{D}$ denote the set of N trajectories collected from expert demonstrations. In our experiments, each trajectory in $\mathcal{D}$ has the same length T, but in general this does not need to be the case.

Learning objective. We begin with the basic imitation learning paradigm of behavioral cloning (Syed and Schapire, 2008). The goal is to learn a policy that behaves like the pre-collected demonstrations:

$$\pi^* = \arg\min_{\pi} \mathbb{E}_{\tau \sim \mathcal{D}} \Big[\mathcal{L}^{\text{imitation}}(\tau, \pi) \Big], \tag{3.1}$$

where $\mathcal{L}^{\text{imitation}}$ is a loss function that quantifies the mismatch between actions chosen by π and those in the demonstrations. Since we are primarily interested in probabilistic or generative policies, we typically use (variants of) negative log-density:

$$\mathcal{L}^{\text{imitation}}(\tau, \pi) = \sum_{t=1}^{T} -\log \pi(\mathbf{a}_t | \mathbf{s}_t), \tag{3.2}$$

where $\pi(\mathbf{a}_t|\mathbf{s}_t)$ is the probability of π picking action $\mathbf{a}_t$ in state $\mathbf{s}_t$.

Policy class of π. Common model choices for instantiating π include sequential generative models like recurrent neural networks (RNN) and trajectory variational autoencoders (TVAE). TVAEs introduce a latent variable $\mathbf{z}$ (also called a trajectory embedding), an encoder network q_ϕ, a policy decoder π_θ, and a prior distribution p

on $\mathbf{z}$. They have been shown to work well in a range of generative policy learning settings (Z. Wang et al., 2017; Ha and Eck, 2018; Co-Reyes et al., 2018), and have the following imitation learning objective:

$$\mathcal{L}^{\text{tvae}}(\tau, \pi_\theta; q_\phi) = \mathbb{E}_{q_\phi(\mathbf{z}|\tau)} \left[\sum_{t=1}^{T} -\log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t, \mathbf{z}) \right] + D_{KL}\big(q_\phi(\mathbf{z}|\tau)||p(\mathbf{z})\big). \quad (3.3)$$

The first term in (3.3) is the standard *negative log-density* that the policy assigns to trajectories in the dataset, while the second term is the *KL-divergence* between the prior and approximate posterior of trajectory embeddings $\mathbf{z}$. The main shortcoming of TVAEs and related approaches, which we address in Sections 3.3 & 3.4, is that the resulting policies cannot be easily calibrated to generate specific styles. For instance, the goal of the trajectory embedding $\mathbf{z}$ is to capture all the styles that exist in the expert demonstrations, but there is no guarantee that the embeddings cleanly encode the desired styles in a calibrated way. Previous work has largely relied on unsupervised learning techniques that either require significant domain knowledge (Le, Yue, et al., 2017), or have trouble scaling to complex styles commonly found in real-world applications (Z. Wang et al., 2017; Yunzhu Li, Song, and Ermon, 2017).

3.3 Programmatic Style-consistency

Building upon the basic setup in Section 3.2, we focus on the setting where the demonstrations $\mathcal{D}$ contain diverse behavior styles. To start, let $\mathbf{y} \in Y$ denote a single style label (e.g., speed or destination, as shown in Figure 3.1). Our goal is to learn a policy π that can be explicitly calibrated to $\mathbf{y}$, i.e., trajectories generated by $\pi(\cdot|\mathbf{y})$ should match the demonstrations in $\mathcal{D}$ that exhibit style $\mathbf{y}$.

Obtaining style labels can be expensive using conventional annotation methods, and unreliable using unsupervised approaches. We instead utilize expert-written programs that can automatically produce style labels. We then formalize a notion of style-consistency as a learning objective, and in Section 3.4 describe a practical learning approach.

Expert-written programs for styles. Inspired by the data programming paradigm (Ratner et al., 2016), expert-written programs programmatically produce weak and noisy labels to learn models on otherwise unlabeled datasets. A significant benefit is that programs are often simple scripts that can be quickly applied to the dataset, which is much cheaper than manual annotations and more reliable than unsupervised methods. In our framework, we study behavior styles that can be represented as

programs λ that map trajectories τ to style labels $\mathbf{y}$. For example:

$$\lambda(\tau) = \mathbb{1}\{\|\mathbf{s}_{T+1} - \mathbf{s}_1\|_2 > c\} \tag{3.4}$$

distinguishes between trajectories with large (greater than a threshold c) versus small total displacement. We experiment with a range of programs, as described in Section 3.5. Many behavior styles used in previous work can be represented as programs, e.g., agent speed (Z. Wang et al., 2017). Multiple programs can be provided at once resulting in a combinatorial space of joint style labels. We use trajectory-level labels $\lambda(\tau)$ in our experiments, but in general, programs can be applied on subsequences $\lambda(\tau_{t:t+h})$ to obtain per-timestep labels, e.g., agent goals (Broll et al., 2019). We can efficiently annotate datasets using programs, which we denote as $\lambda(\mathcal{D}) = \{(\tau_i, \lambda(\tau_i))\}_{i=1}^N$. Our goal can now be phrased as: given $\lambda(\mathcal{D})$, train a policy $\pi : \mathcal{S} \times Y \mapsto \mathcal{A}$ such that $\pi(\cdot|\mathbf{y})$ is calibrated to styles $\mathbf{y}$ found in $\lambda(\mathcal{D})$.

Style-consistency. A key insight in our work is that programs for style labels naturally induce a metric for calibration. If a policy $\pi(\cdot|\mathbf{y})$ is calibrated to λ, we would expect the generated behaviors to be consistent with the label. So, we expect the following loss to be small:

$$\mathbb{E}_{\mathbf{y}\sim p(\mathbf{y}),\tau\sim\pi(\cdot|\mathbf{y})}\left[\mathcal{L}^{\text{style}}\big(\lambda(\tau),\mathbf{y}\big)\right], \tag{3.5}$$

where $p(\mathbf{y})$ is a prior over the style labels, and τ is obtained by executing the style-conditioned policy in the environment. $\mathcal{L}^{\text{style}}$ is thus a disagreement loss over labels that is minimized at $\lambda(\tau) = \mathbf{y}$, e.g., $\mathcal{L}^{\text{style}}\big(\lambda(\tau),\mathbf{y}\big) = \mathbb{1}\{\lambda(\tau) \neq \mathbf{y}\}$ for categorical labels. We refer to (3.5) as the *style-consistency* loss, and say that $\pi(\cdot|\mathbf{y})$ is maximally calibrated to λ when (3.5) is minimized. Our learning objective adds (3.1) with (3.5):

$$\begin{aligned}\pi^* = \arg\min_\pi \mathbb{E}_{\left(\tau,\lambda(\tau)\right)\sim\lambda(\mathcal{D})}&\left[\mathcal{L}^{\text{imitation}}\Big(\tau,\pi\big(\cdot \mid \lambda(\tau)\big)\Big)\right] \\ &+ \mathbb{E}_{\mathbf{y}\sim p(\mathbf{y}),\tau\sim\pi(\cdot|\mathbf{y})}\left[\mathcal{L}^{\text{style}}\big(\lambda(\tau),\mathbf{y}\big)\right].\end{aligned} \tag{3.6}$$

The simplest choice for the prior distribution $p(\mathbf{y})$ is the marginal distribution of styles in $\lambda(\mathcal{D})$. The first term in (3.6) is a standard imitation learning objective and can be tractably estimated using $\lambda(\mathcal{D})$. To enforce style-consistency with the second term, conceptually we need to sample several $\mathbf{y} \sim p(\mathbf{y})$, then several policy rollouts $\tau \sim \pi(\cdot \mid \mathbf{y})$ from the current policy, and query the program for each trajectory.

Furthermore, if λ is a non-differentiable function defined over the entire trajectory, as is the case in (3.4), then we cannot simply backpropagate the style-consistency loss. In Section 3.4, we introduce differentiable approximations to more easily optimize the objective in (3.6).

Combinatorial joint style space. Our notion of style-consistency can be easily extended to optimize for combinatorially-many joint styles when multiple programs are provided. Suppose we have M programs $\{\lambda_i\}_{i=1}^M$ and corresponding label spaces $\{Y_i\}_{i=1}^M$. Let λ denote $(\lambda_1, \ldots, \lambda_M)$ and $\mathbf{y}$ denote $(\mathbf{y}_1, \ldots, \mathbf{y}_M)$. Style-consistency loss becomes:

$$\mathbb{E}_{\mathbf{y}\sim p(\mathbf{y}),\tau\sim\pi(\cdot|\mathbf{y})}\left[\sum_{i=1}^{M}\mathcal{L}_i^{\text{style}}\big(\lambda_i(\tau),\mathbf{y}_i\big)\right]. \tag{3.7}$$

Note that style-consistency is optimal when the generated trajectory agrees with *all* styles. Although challenging to achieve, this outcome is most desirable, i.e. $\pi(\cdot|\mathbf{y})$ is calibrated to *all* styles simultaneously. Indeed, a key metric that we evaluate is how well various learned policies can be calibrated to all styles simultaneously (i.e., loss of 0 only if all styles are calibrated, and loss of 1 otherwise).

3.4 Learning Approach

Optimizing (3.6) is challenging due to the long-time horizon and non-differentiability of the labeling functions λ.[1] Given unlimited queries to the environment, one could naively employ model-free reinforcement learning, e.g., estimating (3.5) using policy rollouts and optimizing using policy gradient approaches. We instead take a model-based approach, described generically in Algorithm 1, that is more computationally-efficient and decomposable (i.e., transparent). The model-based approach is compatible with batch or offline learning, and we found it particularly useful for diagnosing deficiencies in our algorithmic framework. We first introduce a label approximator for λ, and then show how to optimize through the environmental dynamics using a differentiable model-based learning approach.

Approximating programs. To deal with non-differentiability of λ, we approximate it with a differentiable function C_ψ^λ parameterized by ψ:

$$\psi^* = \arg\min_\psi \mathbb{E}_{\left(\tau,\lambda(\tau)\right)\sim\lambda(\mathcal{D})}\Big[\mathcal{L}^{\text{label}}\big(C_\psi^\lambda(\tau),\lambda(\tau)\big)\Big]. \tag{3.8}$$

[1] This issue is not encountered in previous work on style-dependent imitation learning (Yunzhu Li, Song, and Ermon, 2017; Hausman et al., 2017), since they use purely unsupervised methods such as maximizing mutual information, which is differentiable.

Algorithm 1 Generic recipe for optimizing (3.6)

1: **Input**: demonstrations $\mathcal{D}$, programs λ
2: construct $\lambda(\mathcal{D})$ by applying λ on trajectories in $\mathcal{D}$
3: optimize (3.8) to convergence to learn $C^{\lambda}_{\psi^*}$
4: optimize (3.9) to convergence to learn π^*

Here, $\mathcal{L}^{\text{label}}$ is a differentiable loss that approximates $\mathcal{L}^{\text{style}}$, such as cross-entropy loss when $\mathcal{L}^{\text{style}}$ is the $0/1$ loss. In our experiments we use an RNN to represent C^{λ}_{ψ}. We then modify the style-consistency term in (3.6) with $C^{\lambda}_{\psi^*}$ and optimize:

$$\pi^* = \arg\min_{\pi} \mathbb{E}_{\left(\tau,\lambda(\tau)\right)\sim\lambda(\mathcal{D})}\left[\mathcal{L}^{\text{imitation}}\Big(\tau, \pi\big(\cdot \mid \lambda(\tau)\big)\Big)\right] + \mathbb{E}_{\mathbf{y}\sim p(\mathbf{y}),\tau\sim\pi(\cdot|\mathbf{y})}\left[\mathcal{L}^{\text{label}}\big(C^{\lambda}_{\psi^*}(\tau), \mathbf{y}\big)\right]. \tag{3.9}$$

Optimizing $\mathcal{L}^{\text{style}}$ over trajectories. The next challenge is to optimize style-consistency over multiple time steps. Consider the program in (3.4) that computes the difference between the first and last states. Our label approximator $C^{\lambda}_{\psi^*}$ may converge to a solution that ignores all inputs except for $\mathbf{s}_1$ and $\mathbf{s}_{T+1}$. In this case, $C^{\lambda}_{\psi^*}$ provides no learning signal about intermediate steps. As such, effective optimization of style-consistency in (3.9) requires informative learning signals on all actions at every step, which can be viewed as a type of credit assignment problem.

In general, model-free and model-based approaches address this challenge in dramatically different ways and for different problem settings. A model-free solution views this credit assignment challenge as analogous to that faced by reinforcement learning (RL), and repurposes generic reinforcement learning algorithms. Crucially, they assume access to the environment to collect more rollouts under any new policy. A model-based solution does not assume such access and can operate only with the batch of behavior data $\mathcal{D}$; however they can have an additional failure mode since the learned models may provide an inaccurate signal for proper credit assignment. We choose a model-based approach, while exploiting access to the environment when available to refine the learned models, for two reasons: (a) we found it to be compositionally simpler and easier to debug; and (b) we can use the learned model to obtain hallucinated rollouts of any policy efficiently during training.

Modeling dynamics for credit assignment. Our model-based approach utilizes a dynamics model P_{φ} to approximate the environment's dynamics by predicting the

change in state given the current state and action:

$$\varphi^* = \arg\min_{\varphi} \mathbb{E}_{\tau\sim\mathcal{D}} \sum_{t=1}^{T} \mathcal{L}^{\text{dynamics}}\big(P_\varphi(\mathbf{s}_t, \mathbf{a}_t), (\mathbf{s}_{t+1} - \mathbf{s}_t)\big), \tag{3.10}$$

where $\mathcal{L}^{\text{dynamics}}$ is often L_2 or squared-L_2 loss (Nagabandi et al., 2018; Y. Luo et al., 2019). This allows us to generate trajectories by rolling out: $\mathbf{s}_{t+1} = \mathbf{s}_t + P_\varphi\big(\mathbf{s}_t, \pi(\mathbf{s}_t)\big)$. Then optimizing for style-consistency in (3.9) would backpropagate through our dynamics model P_φ and provide informative learning signals to the policy at every timestep.

We outline our model-based approach in Algorithm 2. Lines 10-12 describe an optional step to fine-tune the dynamics model by querying the environment using the current policy (similar to Y. Luo et al. (2019)); we found that this can improve style-consistency in some experiments. In Appendix B.2 we elaborate on how the dynamics model and objective of Eqn (3.10) is changed if the environment is stochastic.

Algorithm 2 Model-based approach for Algorithm 1

1: **Input**: demonstrations $\mathcal{D}$, programs λ, label approximators C^λ_ψ, dynamics P_φ
2: $\lambda(\mathcal{D}) \leftarrow \{\big(\tau_i, \lambda(\tau_i)\big)\}_{i=1}^N$
3: **for** n_{dynamics} iterations **do**
4: optimize (3.10) with batch from $\mathcal{D}$
5: **for** n_{label} iterations **do**
6: optimize (3.8) with batch from $\lambda(\mathcal{D})$
7: **for** n_{policy} iterations **do**
8: $\mathcal{B} \leftarrow$ { n_{collect} trajectories using P_φ and π }
9: optimize (3.9) with batch from $\lambda(\mathcal{D})$ and $\mathcal{B}$
10: **for** n_{env} iterations **do** ▷ Fine-tune P_φ
11: $\tau_{\text{env}} \leftarrow$ { 1 trajectory using environment and π }
12: optimize (3.10) with τ_{env}

Discussion. To summarize, we claim that style-consistency is an objective metric to measure the quality of style-calibration. Our learning approach uses off-the-shelf methods to enforce style-consistency during training. We anticipate several variants of style-consistent policy learning of Algorithm 1, e.g., using model-free RL, using environment/model rollouts to fine-tune the program approximator, using style-conditioned policy classes, or using other loss functions to encourage imitation quality. Our experiments in Section 3.5 establish that our style-consistency

loss provides a clear learning signal, that no prior approach directly enforces this consistency, and that our approach accomplishes calibration for a combinatorial joint style space.

3.5 Experiments

We first briefly describe our experimental setup and baseline choices, and then discuss our main experimental results. A full description of experiments is available in Appendix B.3.

Data. We validate our framework on two datasets: 1) a collection of professional basketball player trajectories with the goal of learning a policy that generates realistic player-movement, and 2) a Cheetah agent running horizontally in MuJoCo (Todorov, Erez, and Tassa, 2012) with the goal of learning a policy with calibrated gaits. The former has a known dynamics function: $P(\mathbf{s}_t, \mathbf{a}_t) = \mathbf{s}_t + \mathbf{a}_t$, where $\mathbf{s}_t$ and $\mathbf{a}_t$ are the player's position and velocity on the court respectively; we expect the dynamics model P_φ to easily recover this function. The latter has an unknown dynamics function (which we learn a model of when approximating style-consistency). We obtain Cheetah demonstrations from a collection of policies trained using `pytorch-a2c-ppo-acktr` (Kostrikov, 2018) to interface with the DeepMind Control Suite's Cheetah domain (Tassa et al., 2018) – see Appendix B.3 for details.

Programs for style labels. Programs for Basketball include: 1) average `Speed` of the player, 2) `Displacement` from initial to final position, 3) distance from final position to a fixed `Destination` on the court (e.g. the basket), 4) mean `Direction` of travel, and 5) `Curvature` of the trajectory, which measures the player's propensity to change directions. For Cheetah, we have programs for the agent's 1) `Speed`, 2) `Torso-height`, 3) `Back-foot-height`, and 4) `Front-foot-height` that can be trivially computed from trajectories extracted from the environment.

We threshold the aforementioned programs into categorical labels (leaving real-valued labels for future work) and use (3.5) for style-consistency with $\mathcal{L}^{\text{style}}$ as the $0/1$ loss. We use cross-entropy for $\mathcal{L}^{\text{label}}$ and list all other hyperparameters in Appendix B.3.

Metrics. We will primarily study two properties of the learned models in our experiments: imitation quality and style-calibration quality. For measuring imitation

quality of generative models, we report the *negative log-density* term in (3.3), also known as the reconstruction loss term in VAE literature (Kingma and Welling, 2014; Ha and Eck, 2018), which corresponds to how well the policy can reconstruct trajectories from the dataset.

To measure style-calibration, we report style-consistency results as $1 - \mathcal{L}^{\text{style}}$ in (3.5) so that all results are easily interpreted as accuracies. In Experiment 5, we find that style-consistency indeed captures a reasonable notion of calibration – when the program is inherently noisy and style-calibration is hard, style-consistency correspondingly decreases. In Experiment 3, we find that the goals of imitation (as measured by negative log-density) and calibration (as measured by style-consistency) may not always be aligned—investigating this trade-off is an avenue for future work.

Baselines. Our main experiments use TVAEs as the underlying policy class. We compare our approach, **CTVAE-style**, with 3 baselines:

1. **CTVAE**: conditional TVAEs (Z. Wang et al., 2017).
2. **CTVAE-info**: CTVAE with information factorization (Creswell, Bharath, and Sengupta, 2017), *indirectly* maximizes style-consistency by removing correlation of $\mathbf{y}$ with $\mathbf{z}$.
3. **CTVAE-mi**: CTVAE with mutual information maximization between style labels and trajectories. This is a supervised variant of unsupervised models (X. Chen et al., 2016; Yunzhu Li, Song, and Ermon, 2017), and also requires learning a dynamics model for sampling policy rollouts.

Detailed descriptions of baselines are in Appendix B.1. All baseline models build upon TVAEs, which are also conditioned on a latent variable (see Section 3.2) and only fundamentally differ in how they encourage the calibration of policies to different style labels. We highlight that the underlying model choice is orthogonal to our contributions; our framework is compatible with other policy models (see Experiment 4 with an RNN policy class).

Model details. We model all trajectory embeddings $\mathbf{z}$ as a diagonal Gaussian with a standard normal prior. Encoder q_ϕ and label approximators C^λ_ψ are bi-directional GRUs (Cho et al., 2014) followed by linear layers. Policy π_θ is recurrent for basketball, but non-recurrent for Cheetah. The Gaussian log sigma returned by π_θ

is state-dependent for basketball, but state-independent for Cheetah. For Cheetah, we made these choices based on prior work in MuJoCo for training gait policies (Z. Wang et al., 2017). For Basketball, we observed a lot more variation in the 500k demonstrations so we experimented with a more flexible model. See Appendix B.3 for hyperparameters.

Experiment 1: How well can we calibrate policies for single styles?

We first threshold programs into 3 classes for Basketball and 2 classes for Cheetah; the marginal distribution $p(\mathbf{y})$ of styles in $\lambda(\mathcal{D})$ is roughly uniform over these classes. Then we learn a policy π^* calibrated to each of these styles. Finally, we generate rollouts from each of the learned policies to measure style-consistency. Table 3.1 compares the median style-consistency (over 5 seeds) of learned policies. For Basketball, CTVAE-style significantly outperforms baselines and achieves almost perfect style-consistency for 4 of the 5 styles. For Cheetah, CTVAE-style outperforms all baselines, but the absolute performance is lower than for Basketball—we conjecture that this is due to the complex environment dynamics that can be challenging for model-based approaches. Figure 3.3 visualizes of our CTVAE-style policy calibrated for `Destination(basket)`.

Model	**Speed**	**Disp.**	**Dest.**	**Dir.**	**Curve**
CTVAE	83	72	82	77	61
CTVAE-info	84	71	79	72	60
CTVAE-mi	86	74	82	77	72
CTVAE-style	**95**	**96**	**97**	**97**	**81**

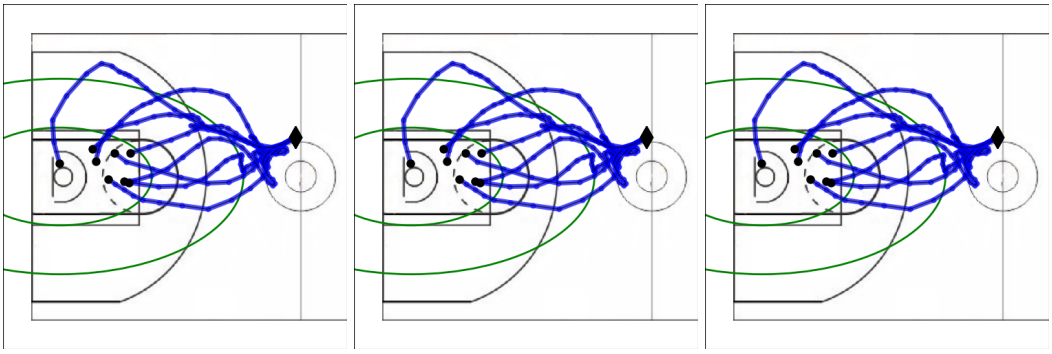
(a) Style-consistency for styles in Basketball, namely `Speed`, `Displacement`, `Destination(basket)`, `Direction`, and `Curvature`.

Model	**Speed**	**Torso-height**	**BFoot-height**	**FFoot-height**
CTVAE	59	63	68	68
CTVAE-info	57	63	65	66
CTVAE-mi	60	65	65	70
CTVAE-style	**79**	**80**	**80**	**77**

(b) Style-consistency for styles in Cheetah, namely `Speed`, `Torso-height`, `Back-foot-height`, and `Front-foot-height`.

Table 3.1: **Individual Style Calibration:** Style-consistency ($\times 10^{-2}$, median over 5 seeds) of policies evaluated with 4,000 Basketball and 500 Cheetah rollouts. Trained separately for each style, CTVAE-style policies outperform baselines for all styles in Basketball and Cheetah environments.

We also consider cases in which programs can have several classes and non-uniform



(a) Label class 0 (slow) (b) Label class 1 (mid) (c) Label class 2 (fast)

Figure 3.3: CTVAE-style rollouts calibrated for `Destination(basket)`, 0.97 style-consistency. Diamonds (♦) and dots (•) indicate initial and final positions. Regions divided by green lines represent label classes.

distributions (i.e., some styles are more/less common than others). We threshold `Displacement` into 6 classes for Basketball and `Speed` into 4 classes for Cheetah and compare the policies in Table 3.2. In general, we observe degradation in overall style-consistency accuracies as the number of classes increase. However, CTVAE-style policies still consistently achieve better style-consistency than baselines in this setting.

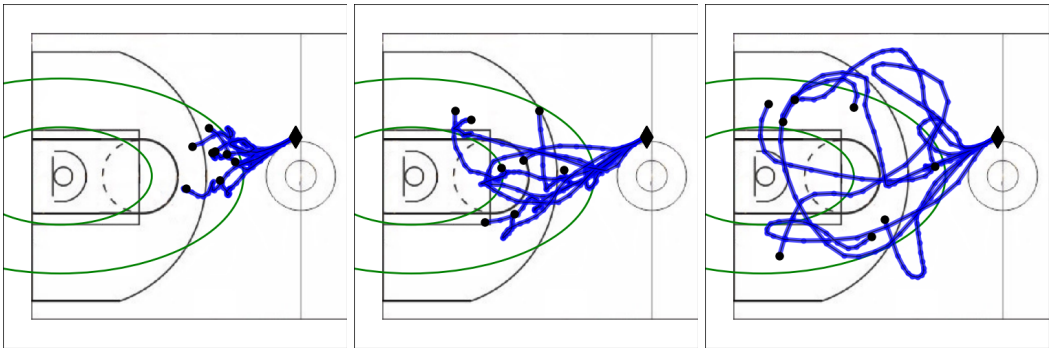
	Basketball				**Cheetah**	
Model	**2 classes**	**3 classes**	**4 classes**	**6 classes**	**3 classes**	**4 classes**
CTVAE	92	83	79	70	45	37
CTVAE-info	90	83	78	70	49	39
CTVAE-mi	92	84	77	70	48	37
CTVAE-style	**99**	**98**	**96**	**92**	**59**	**51**

Table 3.2: **Fine-grained Style-consistency:** ($\times 10^{-2}$, median over 5 seeds) Training on programs with more classes (`Displacement` for Basketball, `Speed` for Cheetah) yields increasingly fine-grained calibration of behavior. Although CTVAE-style degrades as the number of classes increases, it outperforms baselines for all styles.

We visualize and compare policies calibrated for 6 classes of `Displacement` in Figure 3.2. In Figure 3.2b and 3.2d, we see that our CTVAE-policy (0.92 style-consistency) is effectively calibrated for styles of low and high displacement, as all trajectories end in the correct corresponding regions (marked by the green dots). On the other hand, trajectories from a baseline CTVAE model (0.70 style-consistency) in Figure 3.2a and 3.2c can sometimes end in the wrong region corresponding to a different style label (marked by red dots). These results suggest that incorporating

programmatic style-consistency while training via (3.9) can yield good qualitative and quantitative calibration results.

Experiment 2: Can we calibrate for combinatorial joint style spaces?
We now consider combinatorial style-consistency as in (3.7), which measures the style-consistency with respect to *all* programs simultaneously. For instance, in Figure 3.4, we calibrate to both terminating close to the net and also the speed at which the agent moves towards the target destination; if either style is not calibrated then the joint style is not calibrated. In our experiments, we evaluated up to 1024 joint styles.



(a) Label class 0 (slow) (b) Label class 1 (mid) (c) Label class 2 (fast)

Figure 3.4: CTVAE-style rollouts calibrated for 2 styles: label class 1 of `Destination(basket)` in (see Figure 3.3) and each class for `Speed`, with 0.93 style-consistency. Diamonds (⧫) and dots (●) indicate initial and final positions.

Table 3.3 compares the style-consistency of policies simultaneously calibrated for up to 5 programs for Basketball and 3 programs for Cheetah. This is a very challenging task, and we see that style-consistency for baselines degrades significantly as the number of joint styles grows combinatorially. On the other hand, our CTVAE-style approach experiences only a modest decrease in style-consistency and is still significantly better calibrated (0.55 style-consistency vs. 0.21 best baseline style-consistency in the most challenging experiment for Basketball). We visualize a CTVAE-style policy calibrated for two styles in Basketball with style-consistency 0.93 in Figure 3.4. CTVAE-style outperforms baselines in Cheetah as well, but there is still room for improvement to optimize style-consistency better in future work.

Experiment 3: Does style-consistency compromise imitation quality?
In Table 3.4, we investigate whether CTVAE-style's superior style-consistency comes at a significant cost to imitation quality, since we optimize both in (3.6).

Model	**2 styles, 3 classes** (8)	**3 styles, 3 classes** (27)	**4 styles, 3 classes** (81)	**5 styles, 3 classes** (243)	**5 styles, 4 classes** (1024)
CTVAE	71	58	50	37	21
CTVAE-info	69	58	51	32	21
CTVAE-mi	72	56	51	30	21
CTVAE-style	**93**	**88**	**88**	**75**	**55**

(a) Style-consistency for combinatorial styles in Basketball.

Model	**2 styles, 2 classes** (4)	**3 styles, 2 classes** (8)
CTVAE	41	28
CTVAE-info	41	27
CTVAE-mi	40	28
CTVAE-style	**54**	**40**

(b) Style-consistency for combinatorial styles in Cheetah.

Table 3.3: **Combinatorial Style-consistency:** ($\times 10^{-2}$, median over 5 seeds) Simultaneously calibrated to joint styles from multiple programs, CTVAE-style policies significantly outperform all baselines. The number of distinct style combinations are in brackets. The most challenging experiment for basketball calibrates for 1024 joint styles (5 programs, 4 classes each), in which CTVAE-style has a +161% improvement in style-consistency over the best baseline.

For Basketball, high style-consistency is achieved without any degradation in imitation quality. For Cheetah, negative log-density is slightly worse; a followup experiment in Table B.8 in Appendix B.4 shows that we can improve imitation quality with more training, sometimes with modest decrease to style-consistency.

	Basketball		**Cheetah**	
Model	D_{KL}	NLD	D_{KL}	NLD
TVAE	2.55	-7.91	29.4	-0.60
CTVAE	2.51	-7.94	29.3	-0.59
CTVAE-info	2.25	-7.91	29.1	-0.58
CTVAE-mi	2.56	-7.94	28.5	-0.57
CTVAE-style	2.27	-7.83	30.1	-0.28

Table 3.4: KL-divergence and negative log-density per timestep for TVAE models (lower is better). CTVAE-style is comparable to baselines for Basketball, but is slightly worse for Cheetah.

Experiment 4: Is our framework compatible with other policy classes?
We highlight that our framework introduced in Section 3.4 is compatible with any policy class. In this experiment, we optimize for style-consistency using a simpler model for the policy and show that style-consistency is still improved. In particular, we use an RNN and calibrate for `Destination(basket)` in basketball. In Table 3.5, we see that style-consistency is improved for the RNN model without any significant decrease in imitation quality.

	Style-consistency ↑			
Model	**Min**	**Median**	**Max**	NLD ↓
RNN	79	80	81	**-7.7**
RNN-style	**81**	**91**	**98**	-7.6

Table 3.5: Style-consistency of RNN policy model (10^{-2}, 5 seeds) for `Destination(basket)` in basketball. Our approach improves style-consistency without significantly decreasing imitation quality.

Experiment 5: What if programs are noisy?
So far, we have demonstrated that our method optimizing for style-consistency directly can learn policies that are much better calibrated to styles, without a significant degradation in imitation quality. However, we note that the programs used thus far are assumed to be perfect, in that they capture exactly the style that we wish to calibrate. In practice, domain experts may specify programs that are noisy; we simulate that scenario in this experiment.

In particular, we create noisy versions of programs in Table 3.1 by adding Gaussian noise to computed values before applying the thresholds. The noise will result in some label disagreement between noisy and true programs (Table B.12 in Appendix B.4). This resembles the scenario in practice where domain experts can mislabel a trajectory, or have disagreements. We train CTVAE-style models with noisy programs and compute style-consistency using the true programs without noise. Intuitively, we expect the relative decrease in style-consistency to scale linearly with the label disagreement.

Figure 3.5 shows that the median relative decrease in style-consistency of our CTVAE-models scales linearly with label disagreement. Our method is also somewhat robust to noise, as X% label disagreement results in better than X% relative decrease in style-consistency (black line in Figure 3.5). Directions for future work include combining multiple noisy programs together to improve style-consistency

with respect to a "true" program.

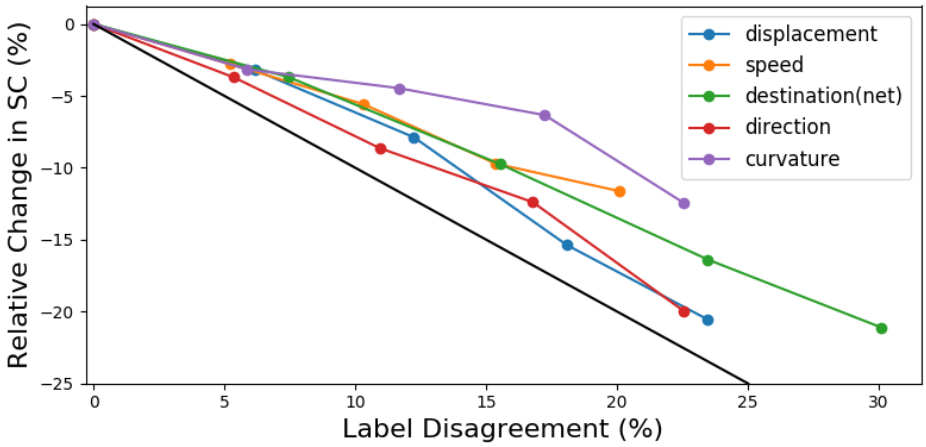


Figure 3.5: Relative change of style-consistency for CTVAE-style policies trained with noisy programs, which are created by injecting noise with mean 0 and standard deviation $c \cdot \sigma$ for $c \in \{1, 2, 3, 4\}$ before applying thresholds to obtain label classes. The x-axis is the label disagreement between noisy and true programs. The y-axis is the median change (5 seeds) in style-consistency using the true programs without noise, relative to Table 3.1. The relationship is generally linear and better than a one-to-one dependency (i.e., if $X\%$ label disagreement leads to $-X\%$ relative change, indicated by the black line). See Table B.12 and B.13 in the Appendix B.4 for more details.

3.6 Related Work

Our work combines ideas from policy learning and data programming to develop a weakly supervised approach for more explicit and fine-grained calibration. As such, our work is related to learning disentangled representations and controllable generative modeling, reviewed below.

Imitation learning of diverse behaviors has focused on unsupervised approaches to infer latent variables/codes that capture behavior styles (Yunzhu Li, Song, and Ermon, 2017; Hausman et al., 2017; Z. Wang et al., 2017). Similar approaches have also been studied for generating text conditioned on attributes such as sentiment or tense (Hu et al., 2017). A typical strategy is to maximize the mutual information between the latent codes and trajectories, in contrast to our notion of programmatic style-consistency.

Disentangled representation learning aims to learn representations where each latent dimension corresponds to exactly one desired factor of variation (Bengio,

Courville, and Vincent, 2012). Recent studies (Locatello et al., 2019) have noted that popular techniques (X. Chen et al., 2016; Higgins et al., 2017; Kim and Mnih, 2018; T. Q. Chen et al., 2018) can be sensitive to hyperparameters and that evaluation metrics can be correlated with certain model classes and datasets, which suggests that fully unsupervised learning approaches may, in general, be unreliable for discovering cleanly calibratable representations. We avoid this roadblock by relying on expert-written programs to provide weak supervision.

Conditional generation for images has recently focused on *attribute manipulation* (Bao et al., 2017; Creswell, Bharath, and Sengupta, 2017; Klys, Snell, and Zemel, 2018), which aims to enforce that changing a label affects only one aspect of the image (similar to disentangled representation learning). We extend these models and compare with our approach in Section 3.5. Our experiments suggest that these algorithms do not necessarily scale well into sequential domains.

Enforcing consistency in generative modeling, such as cycle-consistency in image generation (Zhu et al., 2017), and self-consistency in hierarchical reinforcement learning (Co-Reyes et al., 2018) has proved beneficial. The former minimizes a discriminative disagreement, whereas the latter minimizes a distributional disagreement between two sets of generated behaviors (e.g., KL-divergence). From this perspective, our style-consistency notion is more similar to the former; however, we also enforce consistency over multiple time-steps, which is more similar to the latter.

Goal-conditioned policy learning considers policies that take as input the current state along with a desired goal state (e.g., a location), and then must execute a sequence of actions to achieve the goal states. In some cases, the goal states are provided exogenously (Zheng, Yue, and P. Lucey, 2016; Le, Jiang, et al., 2018; Broll et al., 2019; Ding et al., 2019), and in other cases the goal states are learned as part of a hierarchical policy learning approach (Co-Reyes et al., 2018; Sharma et al., 2020) in a way that uses a self-consistency metric similar to our style-consistency approach. Our approach can be viewed as complementary to these approaches as the goal is to study more general notions of consistency (e.g., our styles subsume goals as a special case) as well as to scale to combinatorial joint style spaces.

Hierarchical control via learning latent motor dynamics is concerned with recovering a latent representation of motor control dynamics such that one can easily design controllers in the latent space (which then get decoded into actions). The high level controllers can then be designed afterwards in a pipelined workflow (Losey et al., 2020; Ling et al., 2020; Y.-S. Luo et al., 2020). The controllers are effective for short time horizons and focus on finding good representations of complex dynamics, whereas we focus on controlling behavior styles that can span longer horizons.

3.7 Discussion

We propose a novel framework for imitating diverse behavior styles while also calibrating to desired styles. Our framework leverages programs to tractably represent styles and introduces programmatic style-consistency, a metric that allows for fair comparison between calibrated policies. Our experiments demonstrate strong empirical calibration results.

We believe that our framework lays the foundation for many directions of future research. First, can one model more complex styles not easily captured with a single program (e.g., aggressive vs. passive play in sports) by composing simpler programs (e.g., max speed, distance to closest opponent, number of fouls committed, etc.), similar to (Ratner et al., 2016; Bach et al., 2017)? Second, can we use these per-timestep labels to model transient styles, or simplify the credit assignment problem when learning to calibrate? Third, can we blend our programmatic supervision with unsupervised learning approaches to arrive at effective semi-supervised solutions? Fourth, can we use model-free approaches to further optimize self-consistency, e.g., to fine-tune from our model-based approach? Finally, can we integrate our framework with reinforcement learning to also optimize for environmental rewards?

References

Bach, Stephen H. et al. (2017). "Learning the Structure of Generative Models without Labeled Data". In: *International Conference on Machine Learning (ICML)*.

Bao, Jianmin et al. (2017). "CVAE-GAN: Fine-Grained Image Generation through Asymmetric Training". In: *IEEE International Conference on Computer Vision (ICCV)*.

Bengio, Yoshua, Aaron C. Courville, and Pascal Vincent (2012). "Unsupervised Feature Learning and Deep Learning: A Review and New Perspectives". In: *arXiv preprint arXiv:1206.5538*.

Bojarski, Mariusz et al. (2016). “End to end learning for self-driving cars”. In: *arXiv preprint arXiv:1604.07316*.

Branson, Kristin et al. (2009). “High-throughput ethomics in large groups of Drosophila”. In: *Nature methods* 6.6, p. 451.

Broll, Brian et al. (2019). “Customizing Scripted Bots: Sample Efficient Imitation Learning for Human-like Behavior in Minecraft”. In: *AAMAS Workshop on Adaptive and Learning Agents*.

Chang, Ming-Fang et al. (2019). “Argoverse: 3D Tracking and Forecasting with Rich Maps”. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Chen, Jianhui et al. (2016). “Learning online smooth predictors for realtime camera planning using recurrent decision trees”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4688–4696.

Chen, Tian Qi et al. (2018). “Isolating Sources of Disentanglement in Variational Autoencoders”. In: *Neural Information Processing Systems (NeurIPS)*.

Chen, Xi et al. (2016). “InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets”. In: *Neural Information Processing Systems (NeurIPS)*.

Cho, Kyunghyun et al. (2014). “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *arXiv preprint arXiv:1406.1078*.

Creswell, Antonia, Anil A. Bharath, and Biswa Sengupta (2017). “Adversarial Information Factorization”. In: *arXiv preprint arXiv:1711.05175*.

Ding, Yiming et al. (2019). “Goal-conditioned imitation learning”. In: *Neural Information Processing Systems (NeurIPS)*.

Eyjolfsdottir, Eyrun, Kristin Branson, et al. (2017). “Learning recurrent representations for hierarchical behavior modeling”. In: *International Conference on Learning Representations*.

Eyjolfsdottir, Eyrun, Steve Branson, et al. (2014). “Detecting social actions of fruit flies”. In: *European Conference on Computer Vision*. Springer, pp. 772–787.

Ha, David and Douglas Eck (2018). “A neural representation of sketch drawings”. In: *International Conference on Learning Representations (ICLR)*.

Hausman, Karol et al. (2017). “Multi-Modal Imitation Learning from Unstructured Demonstrations using Generative Adversarial Nets”. In: *Neural Information Processing Systems (NeurIPS)*.

Higgins, Irina et al. (2017). “beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework”. In: *ICLR*.

Ho, Jonathan and Stefano Ermon (2016). “Generative Adversarial Imitation Learning”. In: *NIPS*.

Hofmann, Katja (2019). "Minecraft as AI Playground and Laboratory". In: *Proceedings of the Annual Symposium on Computer-Human Interaction in Play*, pp. 1–1.

Hu, Zhiting et al. (2017). "Toward Controlled Generation of Text". In: *International Conference on Machine Learning (ICML)*.

Johnson, Matthew J et al. (2016). "Composing graphical models with neural networks for structured representations and fast inference". In: *Advances in Neural Information Processing Systems*.

Kim, Hyunjik and Andriy Mnih (2018). "Disentangling by Factorising". In: *International Conference on Machine Learning (ICML)*.

Kingma, Diederik P and Max Welling (2014). "Auto-encoding variational bayes". In: *International Conference on Learning Representations (ICLR)*.

Klys, Jack, Jake Snell, and Richard S. Zemel (2018). "Learning Latent Subspaces in Variational Autoencoders". In: *Neural Information Processing Systems (NeurIPS)*.

Kostrikov, Ilya (2018). *PyTorch Implementations of Reinforcement Learning Algorithms*. `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail`.

Kurin, Vitaly et al. (2017). "The Atari Grand Challenge Dataset". In: *arXiv preprint arXiv:1705.10998*.

Lample, Guillaume et al. (2017). "Fader networks: Manipulating images by sliding attributes". In: *Neural Information Processing Systems (NeurIPS)*.

Le, Hoang M, Peter Carr, et al. (2017). "Data-driven ghosting using deep imitation learning". In: *MIT Sloan Sports Analytics Conference (SSAC)*.

Le, Hoang M, Nan Jiang, et al. (2018). "Hierarchical imitation and reinforcement learning". In: *International Conference on Machine Learning (ICML)*.

Le, Hoang M, Yisong Yue, et al. (2017). "Coordinated multi-agent imitation learning". In: *International Conference on Machine Learning (ICML)*.

Li, Yaguang et al. (2018). "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting". In: *International Conference on Learning Representations (ICLR)*.

Li, Yunzhu, Jiaming Song, and Stefano Ermon (2017). "InfoGAIL: Interpretable Imitation Learning from Visual Demonstrations". In: *Neural Information Processing Systems (NeurIPS)*.

Ling, Hung Yu et al. (2020). "Character Controllers using Motion VAEs". In: *ACM Conference on Graphics (SIGGRAPH)*.

Locatello, Francesco et al. (2019). "Challenging Common Assumptions in the Unsupervised Learning of Disentangled Representations". In: *International Conference on Machine Learning (ICML)*.

Losey, Dylan P et al. (2020). “Controlling assistive robots with learned latent actions”. In: *International Conference on Robotics and Automation (ICRA)*.

Luo, Wenjie, Bin Yang, and Raquel Urtasun (2018). “Fast and furious: Real time end-to-end 3d detection, tracking and motion forecasting with a single convolutional net”. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Luo, Ying-Sheng et al. (2020). “CARL: Controllable Agent with Reinforcement Learning for Quadruped Locomotion”. In: *ACM Conference on Graphics (SIGGRAPH)*.

Luo, Yuping et al. (2019). “Algorithmic framework for model-based deep reinforcement learning with theoretical guarantees”. In: *International Conference on Learning Representations (ICLR)*.

Nagabandi, Anusha et al. (2018). “Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning”. In: *International Conference on Robotics and Automation (ICRA)*.

Nishimura, Taichi et al. (2019). “Frame Selection for Producing Recipe with Pictures from an Execution Video of a Recipe”. In: *Proceedings of the 11th Workshop on Multimedia for Cooking and Eating Activities*. ACM, pp. 9–16.

Ratner, Alexander et al. (2016). “Data Programming: Creating Large Training Sets, Quickly”. In: *Neural Information Processing Systems (NeurIPS)*.

Co-Reyes, John D et al. (2018). “Self-consistent trajectory autoencoder: Hierarchical reinforcement learning with trajectory embeddings”. In: *International Conference on Machine Learning (ICML)*.

Ross, Stéphane, Geoffrey J. Gordon, and J. Andrew Bagnell (2011). “No-Regret Reductions for Imitation Learning and Structured Prediction”. In: *AISTATS*.

Sharma, Archit et al. (2020). “Dynamics-aware unsupervised discovery of skills”. In: *International Conference on Learning Representations (ICLR)*.

Suwajanakorn, Supasorn, Steven M Seitz, and Ira Kemelmacher-Shlizerman (2017). “Synthesizing obama: learning lip sync from audio”. In: *ACM Transactions on Graphics (TOG)* 36.4, p. 95.

Syed, Umar and Robert E Schapire (2008). “A game-theoretic approach to apprenticeship learning”. In: *NIPS*.

Tassa, Yuval et al. (2018). “DeepMind Control Suite”. In: *arXiv preprint arXiv:1801.00690*.

Taylor, Sarah et al. (2017). “A deep learning approach for generalized speech animation”. In: *SIGGRAPH*.

Todorov, Emanuel, Tom Erez, and Yuval Tassa (2012). “MuJoCo: A physics engine for model-based control”. In: *International Conference on Intelligent Robots and Systems (IROS)*.

Wang, Ziyu et al. (2017). "Robust Imitation of Diverse Behaviors". In: *Neural Information Processing Systems (NeurIPS)*.

Yeh, Raymond A et al. (2019). "Diverse generation for multi-agent sports games". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Zhan, Eric, Albert Tseng, et al. (2020). "Learning Calibratable Policies using Programmatic Style-Consistency". In: *International Conference on Machine Learning*. PMLR, pp. 11001–11011. URL: https://proceedings.mlr.press/v119/zhan20a.html.

Zhan, Eric, Stephan Zheng, et al. (2019). "Generating Multi-Agent Trajectories using Programmatic Weak Supervision". In: *International Conference on Learning Representations (ICLR)*.

Zheng, Stephan, Yisong Yue, and Patrick Lucey (2016). "Generating long-term trajectories using deep hierarchical networks". In: *Advances in Neural Information Processing Systems*.

Zhu, Jun-Yan et al. (2017). "Unpaired image-to-image translation using cycle-consistent adversarial networks". In: *Proceedings of the IEEE international conference on computer vision*, pp. 2223–2232.

Chapter 4

LEARNING DIFFERENTIABLE NEUROSYMBOLIC PROGRAMS

The work in this chapter was published in (Shah et al., 2020). E.Z. formulated and implemented the method, conducted experiments and analyzed results, and participated in the writing of the manuscript.

Summary

We study the problem of learning differentiable functions expressed as programs in a domain-specific language. Such programmatic models can offer benefits such as composability and interpretability; however, learning them requires optimizing over a combinatorial space of program "architectures". We frame this optimization problem as a search in a weighted graph whose paths encode top-down derivations of program syntax. Our key innovation is to view various classes of neural networks as continuous relaxations over the space of programs, which can then be used to complete any partial program. This relaxed program is differentiable and can be trained end-to-end, and the resulting training loss is an approximately admissible heuristic that can guide the combinatorial search. We instantiate our approach on top of A* search and an iteratively deepened branch-and-bound search, and use these algorithms to learn programmatic classifiers in three sequence classification tasks. Our experiments show that the algorithms outperform state-of-the-art methods for program learning, and that they discover programmatic classifiers that yield natural interpretations and achieve competitive accuracy.

4.1 Introduction

An emerging body of work advocates *program synthesis* as an approach to machine learning. These methods learn functions represented as programs in symbolic, domain-specific languages (DSLs) (Ellis, Solar-Lezama, and J. Tenenbaum, 2016; Ellis, Ritchie, et al., 2018; Young, Bastani, and Naik, 2019; Valkov et al., 2018; Verma, Murali, et al., 2018; Verma, H. M. Le, et al., 2019). Such symbolic models have a number of appeals: they can be more interpretable than neural models, they use the inductive bias embodied in the DSL to learn reliably, and they use compositional language primitives to transfer knowledge across tasks.

In this chapter, we study how to learn *differentiable* programs, which use structured, symbolic primitives to compose a set of parameterized, differentiable modules. Differentiable programs have recently attracted much interest due to their ability to leverage the complementary advantages of programming language abstractions and differentiable learning. For example, recent work has used such programs to compactly describe modular neural networks that operate over rich, recursive data types (Valkov et al., 2018).

To learn a differentiable program, one needs to induce the program's "architecture" while simultaneously optimizing the parameters of the program's modules. This co-design task is difficult because the space of architectures is combinatorial and explodes rapidly. Prior work has approached this challenge using methods such as greedy enumeration, Monte Carlo sampling, Monte Carlo tree search, and evolutionary algorithms (Verma, Murali, et al., 2018; Valkov et al., 2018; Ellis, Nye, et al., 2019). However, such approaches can often be expensive, due to not fully exploiting the structure of the underlying combinatorial search problem.

In this chapter, we show that the differentiability of programs opens up a new line of attack on this search problem. A standard strategy for combinatorial optimization is to exploit (ideally fairly tight) continuous relaxations of the search space (Pearl, 1984; Charniak and Husain, 1991; Xiang and Kim, 2013; Sontag et al., 2012; Korf, 2000; Bagchi and Mahanti, 1983; Verma, H. M. Le, et al., 2019). Optimization in the relaxed space is typically easier and can efficiently guide search algorithms towards good or optimal solutions. In the case of program learning, we propose to use various classes of neural networks as relaxations of partial programs. We frame our problem as searching a graph, in which nodes encode program architectures with missing expressions, and paths encode top-down program derivations. For each partial architecture u encountered during this search, the relaxation amounts to substituting the unknown part of u with a neural network with free parameters. Because programs are differentiable, this network can be trained on the problem's end-to-end loss. If the space of neural networks is an (approximate) proper relaxation of the space of programs (and training identifies a near-optimum neural network), then the training loss for the relaxation can be viewed as an (approximately) admissible heuristic.

We instantiate our approach, called NEAR (abbreviation for ***Ne**ural **A**dmissible **R**elaxation*), on top of two informed search algorithms: A* search and an iteratively deepened depth-first search that uses a heuristic to direct branching as well as branch-and-bound pruning (IDS-BB). We evaluate the algorithms in the task of

learning programmatic classifiers in three behavior classification applications. We show that the algorithms substantially outperform state-of-the-art methods for program learning, and can learn classifier programs that bear natural interpretations and are close to neural models in accuracy.

To summarize, this chapter makes three contributions. First, we identify a tool—heuristics obtained by training neural relaxations of programs—for accelerating combinatorial searches over differentiable programs. As far as we know, this is the first approach to exploit the differentiability of a programming language in program synthesis. Second, we instantiate this idea using two classic search algorithms. Third, we present promising experimental results in three sequence classification applications.

4.2 Problem Formulation

We view a program in our domain-specific language (DSL) as a pair (α, θ), where α is a discrete *(program) architecture* and θ is a vector of real-valued parameters. The architecture α is generated using a *context-free grammar* (Hopcroft, Motwani, and Ullman, 2007). The grammar consists of a set of rules $X \to \sigma_1 \ldots \sigma_k$, where X is a *nonterminal* and $\sigma_1, \ldots, \sigma_k$ are either nonterminals or *terminals*. A nonterminal stands for a missing subexpression; a terminal is a symbol that can actually appear in a program's code. The grammar starts with an initial nonterminal, then iteratively applies the rules to produce a series of *partial architectures*: programs made from one or more nonterminals and zero or more terminals. The process continues until there are no nonterminals left, i.e., we have a complete architecture.

The semantics of the architecture α is given by a function $[[\alpha]](x, \theta)$, defined by rules that are fixed for the DSL. We require this function to be differentiable in θ. Also, we define a *structural cost* for architectures. Let each rule r in the DSL grammar have a non-negative real cost $s(r)$. The structural cost of α is $s(\alpha) = \sum_{r \in \mathcal{R}(\alpha)} s(r)$, where $\mathcal{R}(\alpha)$ is the multiset of rules used to create α. Intuitively, architectures with lower structural cost are simpler are more human-interpretable.

To define our learning problem, we assume an unknown distribution $D(x, y)$ over inputs x and labels y, and consider the prediction error function $\zeta(\alpha, \theta) = \mathbb{E}_{(x,y)\sim D}[\mathbb{1}\{[[\alpha]](x, \theta) \neq y\}]$, where $\mathbb{1}$ is the indicator function. Our goal is to find an architecturally simple program with low prediction error, i.e., to solve the

optimization problem:

$$(\alpha^*, \theta^*) = \arg\min_{(\alpha,\theta)}(s(\alpha) + \zeta(\alpha, \theta)). \tag{4.1}$$

Program Learning for Sequence Classification.

Program learning is applicable in many settings; we specifically study it in the sequence classification context (Dietterich, 2002). We now sketch our DSL for this domain. Like many others DSLs for program synthesis (Feser, Chaudhuri, and Dillig, 2015; Balog et al., 2017; Valkov et al., 2018), our DSL is purely functional. The language has the following characteristics:

- Programs in the DSL operate over two data types: real vectors and sequences of real vectors. We assume a simple type system that makes sure that these types are used consistently.
- Programs use a set of fixed algebraic operations $\oplus$ as well as a "library" of differentiable, parameterized functions $\oplus_\theta$. Because we are motivated by interpretability, the library used in our current implementation only contains affine transformations. In principle, it could be extended to include other kinds of functions as well.
- Programs use a set of higher-order combinators to recurse over sequences. In particular, we allow the standard **map** and **fold** combinators. To compactly express sequence-to-sequence functions, we also allow a special **mapprefix** combinator. Let e be a function that maps sequences to vectors. For a sequence x, **mapprefix**(e, x) equals the sequence $\langle e(x_{[1:1]}), e(x_{[1:2]}), \ldots, e(x_{[1:n]})\rangle$, where $x_{[1:i]}$ is the i-th prefix of x.
- Programs can use a conditional branching construct. However, to avoid discontinuities, we interpret this construct in terms of a smooth approximation:

$$\begin{aligned}&[[\textbf{if } \alpha_1 > 0 \textbf{ then } \alpha_2 \textbf{ else } \alpha_3]](x, (\theta_1, \theta_2, \theta_3))\\ &\quad = \sigma(\beta \cdot [[\alpha_1]](x, \theta_1)) \cdot [[\alpha_2]](x, \theta_2)\\ &\qquad + (1 - \sigma(\beta \cdot [[\alpha_1]](x, \theta_1))) \cdot [[\alpha_3]](x, \theta_3).\end{aligned} \tag{4.2}$$

  Here, σ is the sigmoid function and β is a temperature hyperparameter. As $\beta \to 0$, this approximation approaches the usual if-then-else construct.

Figure 4.1 summarizes our DSL in the standard Backus-Naur form (Winskel, 1993). Figures 4.2 and 4.3 show two programs synthesized by our learning procedure

using our DSL with libraries of domain-specific affine transformations. Both programs offer an interpretation in their respective domains, while offering respectable performance against an RNN baseline.

$$\begin{aligned}\alpha \quad ::= \quad & x \mid c \mid \oplus(\alpha_1, \ldots, \alpha_k) \mid \oplus_\theta(\alpha_1, \ldots, \alpha_k) \mid \textbf{if } \alpha_1 \textbf{ then } \alpha_2 \textbf{ else } \alpha_3 \mid \textbf{sel}_S\ x \\ & \textbf{map}(\alpha_1,\ x) \mid \textbf{fold}(\alpha_1,\ c,\ x) \mid \textbf{mapprefix}(\alpha_1,\ x)\end{aligned}$$

Figure 4.1: Grammar of DSL for sequence classification. Here, x, c, $\oplus$, and $\oplus_\theta$ represent inputs, constants, basic algebraic operations, and parameterized library functions, respectively. $\textbf{sel}_S$ returns a vector consisting of a subset S of the dimensions of an input x.

$$\begin{aligned}&\textbf{map}(\\ &\quad \textbf{if } \texttt{DistAffine}_{[.0217];-.2785}(\mathbf{s}_t)\\ &\quad\quad \textbf{then } \texttt{AccelAffine}_{[-.0007,.0055,.0051,-.0025];3.7426}(\mathbf{s}_t)\\ &\quad \textbf{else } \texttt{DistAffine}_{[-.2143];1.822)}(\mathbf{s}_t),\\ &\tau)\end{aligned}$$

Figure 4.2: Synthesized program classifying a "sniff" action between two mice in the CRIM13 dataset. `DistAffine` and `AccelAffine` are functions that first select the parts of the input states $\mathbf{s}_t$ that represent distance and acceleration measurements, respectively, and then apply affine transformations to the resulting vectors. In the parameters (subscripts) of these functions, the brackets contain the weight vectors for the affine transformation, and the succeeding values are the biases. The program achieves an accuracy of 0.87 (vs. 0.89 for RNN baseline) and can be interpreted as follows: if the distance between two mice is small, they are doing a "sniff" (large bias in **else** clause). Otherwise, they are doing a "sniff" if the difference between their accelerations is small.

4.3 Program Learning using NEAR

We formulate our program learning problem as a form of graph search. The search derives program architectures top-down: it begins with the *empty* architecture, generates a series of partial architectures following the DSL grammar, and terminates when a complete architecture is derived.

In more detail, we imagine a graph $\mathcal{G}$ in which:

- The node set consists of all partial and complete architectures permissible in the DSL. Let u denote partial architectures (at least one nonterminal), and α, as before, denote complete architectures (no nonterminals).

map(
  multiply(
    add(`OffenseAffine`($\mathbf{s}_t$), `BallAffine`($\mathbf{s}_t$)),
    add(`OffenseAffine`($\mathbf{s}_t$), `BallAffine`($\mathbf{s}_t$))
  ),
τ)

Figure 4.3: Synthesized program classifying the ballhandler for basketball. `OffenseAffine` and `BallAffine` are parameterized affine transformations over the xy-coordinates of the offensive players and the ball. **multiply** and **add** are computed element-wise. The program structure can be interpreted as computing the Euclidean norm/distance between the offensive players and the ball and suggests that this quantity can be important for determining the ballhandler. On a set of learned parameters (not shown), this program achieves an accuracy of 0.905 (vs. 0.945 for an RNN baseline).

- The *source node* u_0 is the empty architecture, while complete architectures α are *goal nodes*.
- Edges are directed and capture single-step applications of rules of the DSL. Edges can be divided into: (i) *internal edges* (u, u') between partial architectures u and u', and (ii) *goal edges* (u, α) between partial architecture u and complete architecture α. An internal edge (u, u') exists if one can obtain u' by substituting a nonterminal in u following a rule of the DSL. A goal edge (u, α) exists if we can complete u into α by applying a rule r of the DSL.
- The cost of an internal edge (u, u') is given by the structural cost $s(r)$, where r is the rule used to construct u' from u. The cost of a goal edge (u, α) is $s(r) + \zeta(\alpha, \theta^*)$, where $\theta^* = \arg\min_\theta \zeta(\alpha, \theta)$ and r is the rule used to construct α from u.

A path in the graph $\mathcal{G}$ is defined as usual, as a sequence of nodes $u_1, \ldots, u_k$ such that there is an edge (u_i, u_{i+1}) for each $i \in \{1, \ldots, k-1\}$. The cost of a path is the sum of the costs of these edges. Our goal is to discover a least-cost path from the source u_0 to some goal node α^*. Then by construction of our edge costs, α^* is an optimal solution to our learning problem in (4.1).

Neural Relaxations as Admissible Heuristics

The main challenge in our search problem is that our goal edges contain rich cost information, but this information is only accessible when a path has been explored until the end. A heuristic function $h(u)$ that can predict the value of choices made at nodes u encountered early in the search can help with this difficulty. If such a heuristic is *admissible*—i.e., underestimates the cost-to-go—it enables the use of informed search strategies such as A* and branch-and-bound while guaranteeing optimal solutions. Our NEAR approach (abbreviation for ***Ne**ural **A**dmissible **R**elaxation*) uses neural approximations of spaces of programs to construct a heuristic that is ϵ-close to being admissible.

Let a *completion* of a partial architecture u be a (complete) architecture $u[\alpha_1, \ldots, \alpha_k]$ obtained by replacing the nonterminals in u by suitably typed architectures α_i. Let θ_u be the parameters of u and θ be parameters of all α_i. The *cost-to-go* at u is given by:

$$J(u) = \min_{\alpha_1,\ldots,\alpha_k,\theta_u,\theta} ((s(u[\alpha_1, \ldots, \alpha_k] - s(u)) + \zeta(u[\alpha_1, \ldots, \alpha_k], (\theta_u, \theta)), \quad (4.3)$$

where the structural cost $s(u)$ is the sum of the costs of the grammatical rules used to construct u.

To compute a heuristic cost $h(u)$ for a partial architecture u encountered during search, we substitute the nonterminals in u with neural networks parameterized by ω. These networks are *type-correct*—for example, if a nonterminal is supposed to generate subexpressions whose inputs are sequences, then the neural network used in its place is recurrent. We show an example of NEAR used in a program learning-graph search formulation in Figure 4.4.

We view the *neurosymbolic* programs resulting from this substitution as tuples $(u, (\theta_u, \omega))$. We define a semantics for such programs by extending our DSL's semantics, and lift the function ζ to assign costs $\zeta(u, (\theta_u, \omega))$ to such programs. The heuristic cost for u is now given by:

$$h(u) = \min_{\theta_u,\omega} \zeta(u, (\theta_u, \omega)). \quad (4.4)$$

As $\zeta(u, (\theta_u, \omega))$ is differentiable in θ_u and ω, we can compute $h(u)$ using gradient descent.

ϵ-Admissibility. In practice, the neural networks that we use may only form an approximate relaxation of the space of completions and parameters of architectures;

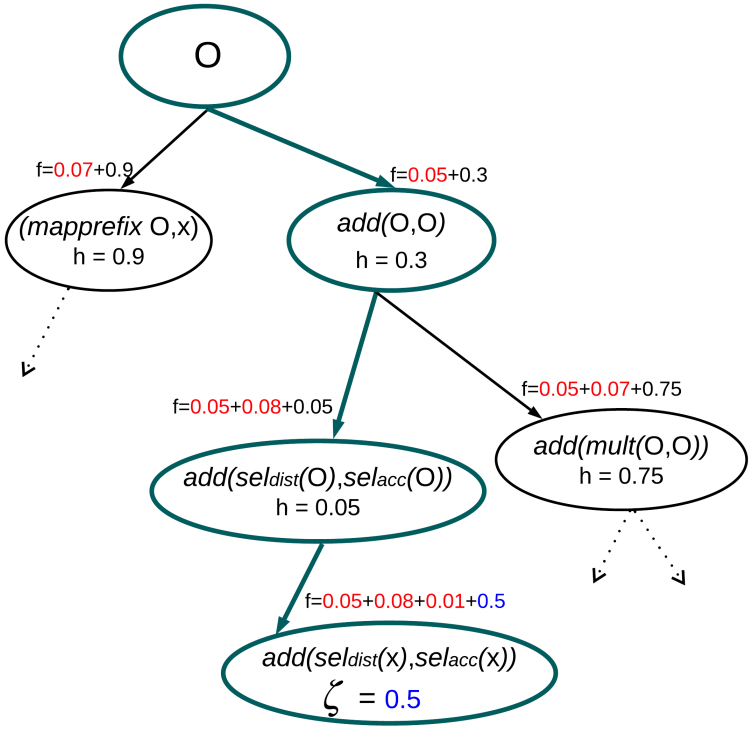


Figure 4.4: An example of program learning formulated as graph search. Structural costs are in red, heuristic values in black, prediction errors ζ in blue, O refers to a nonterminal in a partial architecture, and the path to a goal node returned by A*-NEAR search is in teal.

also, the training of these networks may not reach global optima. To account for these errors, we consider an approximate notion of admissibility. Many such notions have been considered in the past (Harris, 1974; Pearl, 1984; Valenzano et al., 2013); here, we follow a definition used by (Harris, 1974).

For a fixed constant $\epsilon > 0$, let an *ϵ-admissible heuristic* be a function $h^*(u)$ over architectures such that $h^*(u) \leq J(u) + \epsilon$ for all u. Now consider any completion $u[\alpha_1, \ldots, \alpha_k]$ of an architecture u. As neural networks with adequate capacity are universal function approximators, there exist parameters ω^* for our neurosymbolic program such that for all $u, \alpha_1, \ldots, \alpha_k, \theta_u$, and θ:

$$\zeta(u, (\theta_u, \omega^*)) \leq \zeta(u[\alpha_1, \ldots, \alpha_k], (\theta_u, \theta)) + \epsilon. \tag{4.5}$$

Because edges in our search graph have non-negative costs, $s(u) \leq s(u[\alpha_1, \ldots, \alpha_k])$,

implying:

$$\begin{aligned} h(u) &\leq \min_{\alpha_1,\ldots,\alpha_k,\theta_u,\theta} \zeta(u[\alpha_1,\ldots,\alpha_k],(\theta_u,\theta)) + \epsilon \\ &\leq \min_{\alpha_1,\ldots,\alpha_k,\theta_u,\theta} \zeta(u[\alpha_1,\ldots,\alpha_k],(\theta_u,\theta)) + (s(u[\alpha_1,\ldots,\alpha_k]) - s(u)) + \epsilon \\ &= J(u) + \epsilon. \end{aligned} \tag{4.6}$$

In other words, $h(u)$ is ϵ-admissible.

Empirical Considerations. We have formulated our learning problem in terms of the true prediction error $\zeta(\alpha, \theta)$. In practice, we must use statistical estimates of this error. Following standard practice, we use an empirical validation error to choose architectures, and an empirical training error is used to choose module parameters. This means that in practice, the cost of a goal edge (u, α) in our graph is $\zeta^{\text{val}}(\alpha, \arg\min_\theta \zeta^{\text{train}}(\alpha, \theta))$.

One complication here is that our neural heuristics encode both the completions of an architecture and the parameters of these completions. Training a heuristic on either the training loss or the validation loss will introduce an additional error. Using standard generalization bounds, we can argue that for adequately large training and validation sets, this error is bounded (with probability arbitrarily close to 1) in either case, and that our heuristic is ϵ-admissible with high probability in spite of this error.

Integrating NEAR with Graph Search Algorithms

The NEAR approach can be used in conjunction with any heuristic search algorithm (Russell and Norvig, 2002) over architectures. Specifically, we have integrated NEAR with two classic graph search algorithms: A* (Pearl, 1984) (Algorithm 3) and an iteratively deepened depth-first search with branch-and-bound pruning (IDS-BB) (Algorithm 8 in Appendix C.1). Both algorithms maintain a *search frontier* by computing an f*-score* for each node: $f(u) = g(u) + h(u)$, where $g(u)$ is the incurred path cost from the source node u_0 to the current node u, and $h(u)$ is a heuristic estimate of the cost-to-go from node u. Additionally, IDS-BB prunes nodes from the frontier that have a higher f-score than the minimum path cost to a goal node found so far.

ϵ-Optimality. An important property of a search algorithm is *optimality*: when multiple solutions exist, the algorithm finds an optimal solution. Both A* and IDS-BB

Algorithm 3 A* search

```
1: Input: Graph G with source u_0
2: S := {u_0}, f(u_0) := ∞
3: while S ≠ ∅ do
4:     v := arg min_{u∈S} f(u)
5:     S := S \ {v}
6:     if v is a goal node then
7:         return v, f_v
8:     else
9:         for child w of v do
10:            Compute g(w), h(w), f(w)
11:            S := S ∪ {w}
```

are optimal given admissible heuristics. An argument by (Harris, 1974) shows that under heuristics that are ϵ-admissible in our case, the algorithms return solutions that at most an additive constant ϵ away from the optimal solution. Let C^* denote the optimal path cost in our graph $\mathcal{G}$, and let $h(u)$ be an ϵ-admissible heuristic (Eq. (4.6)). Suppose IDS-BB or A* returns a goal node α_G that does not have the optimal path cost C^*. Then there must exist a node u_O on the frontier that lies along the optimal path and has yet to be expanded. This lets us establish an upper bound on the path cost of α_G:

$$\begin{aligned} g(\alpha_G) &= f(\alpha_G) \\ &\leq f(u_O) \\ &= g(u_O) + h(u_O) \\ &\leq g(u_O) + J(u_O) + \epsilon \\ &\leq C^* + \epsilon. \end{aligned} \tag{4.7}$$

This line of reasoning can also be extended to the branch-and-bound component of the NEAR-guided IDS-BB algorithm. Consider encountering a goal node during search that sets the branch-and-bound upper threshold to be a cost C. In the remainder of search, some node u_p with an f-cost greater than C is pruned, and the optimal path from u_p to a goal node will not be searched. Assuming the heuristic function h is ϵ-admissible, we can set a lower bound on the optimal path cost from

u_p, $f(u_p^*)$, to be $C - \epsilon$ by the following:

$$\begin{aligned} f(u_p^*) &= g(u_p) + J(u_p) \\ &\geq f(u_p) \\ &= g(u_p) + h(u_p) + \epsilon \\ &\geq g(u_p) + h(u_p) \\ &= C \\ &\geq C - \epsilon. \end{aligned} \tag{4.8}$$

Thus, the IDS-BB algorithm will find goal paths are at worst an additive factor of ϵ more than any pruned goal path.

4.4 Experiments

Datasets for Sequence Classification

For all datasets below, we augment the base DSL in Figure 4.1 with domain-specific library functions that include 1) learned affine transformations over a subset of features, and 2) sliding window feature-averaging functions. Trajectories are represented as sequences of states: $\tau = \{\mathbf{s}_1, \mathbf{s}_2, \dots\}$.

CRIM13. The *CRIM13* dataset (Burgos-Artizzu et al., 2012) contains trajectories for a pair of mice engaging in social behaviors, annotated for different actions per frame by behavior experts; we aim to learn programs for classifying actions at each frame for fixed-size trajectories. Each frame is represented by a 19-dimensional state vector: 4 features capture the xy-positions of the mice, and the remaining 15 features are derived from the positions, such as velocities and distance between mice. We learn programs for two actions that can be identified the tracking features: "sniff" and "other" ("other" is used when there is no behavior of interest occurring). We cut every 100 frames as a trajectory, and in total we have 12,404 training, 3,077 validation, and 2,953 test trajectories.

Fly-vs.-Fly. We use the *Aggression* and *Boy-meets-Boy* datasets within the *Fly-vs.-Fly* environment that tracks a pair of fruit flies and their actions as they interact in different contexts (Eyjolfsdottir et al., 2014). We aim to learn programs that classify trajectories as one of 7 possible actions displaying aggressive, threatening, and nonthreatening behaviors. The length of trajectories can range from 1 to over 10,000 timesteps, but we segment the data into trajectories with a maximum length of 300 for computational efficiency. The average length of a trajectory in our

training set is 42.06 timesteps. We have 5,339 training, 594 validation, and 1,048 test trajectories.

Basketball. We use a subset of the basketball dataset from (Yue et al., 2014) that tracks the movements of professional basketball players. Each trajectory is of length 25 and contains the xy-positions of 5 offensive players, 5 defensive players, and the ball (22 state features per frame). We aim to learn programs that can predict which offensive player has the ball (the "ballhandler") or whether the ball is being passed. In total, we have 18,000 training, 2,801 validation, and 2,693 test trajectories.

Overview of Baseline Program Learning Strategies

We compare our NEAR-guided graph search algorithms, A*-NEAR and IDS-BB-NEAR, with four baseline program learning strategies: 1) top-down enumeration, 2) Monte-Carlo sampling, 3) Monte-Carlo tree search, and 4) a genetic algorithm. We also compare the performance of these program learning algorithms with an RNN baseline (1-layer LSTM).

1) Top-down enumeration. We synthesize and evaluate complete programs in order of increasing complexity measured using the structural cost $s(\alpha)$. This strategy is widely employed in program learning contexts (Valkov et al., 2018; Verma, Murali, et al., 2018; Verma, H. M. Le, et al., 2019) and is provably complete. Since our graph $\mathcal{G}$ grows infinitely, our implementation is akin to breadth-first search up to a specified depth.

2) Monte-Carlo (MC) sampling. Starting from the source node u_0, we sample complete programs by sampling rules (edges) with probabilities proportional to their structural costs $s(r)$. The next node chosen along a path has the best average performance of samples that descended from that node. We repeat the procedure until we reach a goal node and return the best program found among all samples.

3) Monte-Carlo tree search (MCTS). Starting from the source node u_0, we traverse the graph until we reach a complete program using the UCT selection criteria (Kocsis and Szepesvári, 2006), where the value of a node is inversely proportional to the cost of its children. In the backpropagation step we update the value of all nodes along the path. After some iterations, we choose the next node in the path with the highest value. We repeat the procedure until we reach a goal node and return the best program found.

4) Genetic algorithm. We follow the formulation in (Valkov et al., 2018). In our genetic algorithm, crossover, selection, and mutation operations evolve a population of programs over a number of generations until a predetermined number of programs have been trained. The crossover and mutation operations only occur when the resulting program is guaranteed to be type-safe.

For all baseline algorithms, as well as A*-NEAR and IDS-BB-NEAR, model parameters (θ) were learned with the training set, whereas program architectures (α) were evaluated using the performance on the validation set. Additionally, all baselines (including NEAR algorithms) used F1-score (Sasaki, 2007) error as the evaluation objective ζ with which programs were chosen. To account for class imbalances, F1-scoring is commonly used as an evaluation metric in behavioral classification domains, such as those considered in this work (Eyjolfsdottir et al., 2014; Burgos-Artizzu et al., 2012).

Experimental Results

Performance of learned programs. Table 4.1 shows the performance results on the test sets of our program learning algorithms, averaged over 3 trials. The same structural cost function $s(\alpha)$ is used for all algorithms, but can vary across domains. Our NEAR-guided search algorithms consistently outperform other baselines in F1-score while accuracy is comparable (note that our ζ does not include accuracy). Furthermore, NEAR-guided search algorithms are capable of finding deeper and more complex programs that can offer non-trivial interpretations, such as the ones shown in Figures 4.2 and 4.3. Lastly, we verify that our learned programs are comparable with highly expressive RNNs, and see that there is at most a 10% drop in F1-score when using NEAR-guided search algorithms with our DSL.

Efficiency of NEAR-guided graph search. Figure 4.5 tracks the progress of each program learning algorithm by following the median best path cost (4.1) found at a given time across 3 independent trials. Algorithms for each domain were run on the same machine to ensure consistency, and each non-NEAR baseline was set up such to have at least as much time as our NEAR-guided algorithms for their search procedures (see Appendix C.2). We observe that NEAR-guided search algorithms are able to find low-cost solutions more efficiently than existing baselines, while maintaining an overall shorter running time.

	CRIM13-sniff			CRIM13-other			Fly-vs.-Fly			Basketball		
	Acc.	F1	d	Acc.	F1	d	Acc.	F1	d	Acc.	F1	d
Enum.	.851	.221	3	.707	.762	2	.819	.863	2	.844	.857	6.3
MC	.843	.281	7	.630	.715	1	.833	.852	4	.841	.853	6
MCTS	.745	.338	8.7	.666	.749	1	.817	.857	4.7	.711	.729	8
Genetic	.829	.181	1.7	.727	.768	3	.850	.868	6	.843	.853	6.7
IDS-BB-NEAR	.829	.446	6	.729	.768	1.3	.876	.892	4	.889	.903	8
A*-NEAR	.821	.369	6	.706	.764	2.7	.872	.885	4	.906	.918	8
RNN	.889	.481	-	.756	.785	-	.963	.964	-	.945	.950	-

Table 4.1: Mean accuracy, F1-score, and program depth d of learned programs (3 trials). Programs found using our NEAR algorithms consistently achieve better F1-score than baselines and match more closely to the RNN's performance. Our algorithms are also able to search and find programs of much greater depth than the baselines. Experiment hyperparameters are included in Appendix C.2.

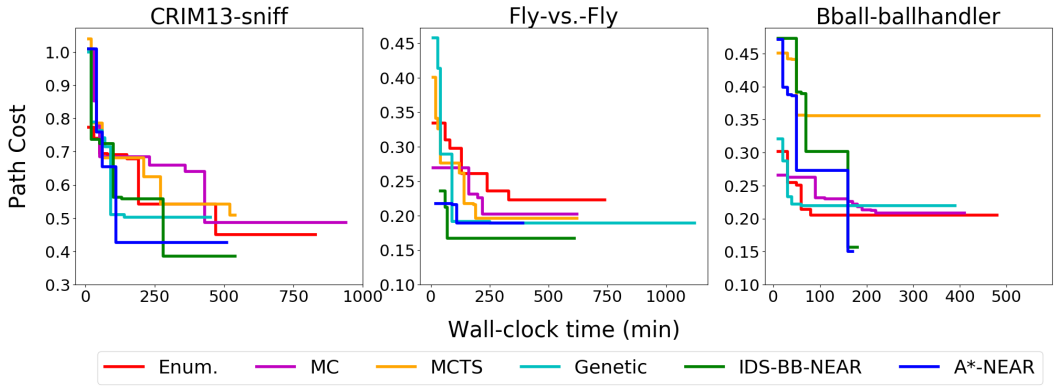


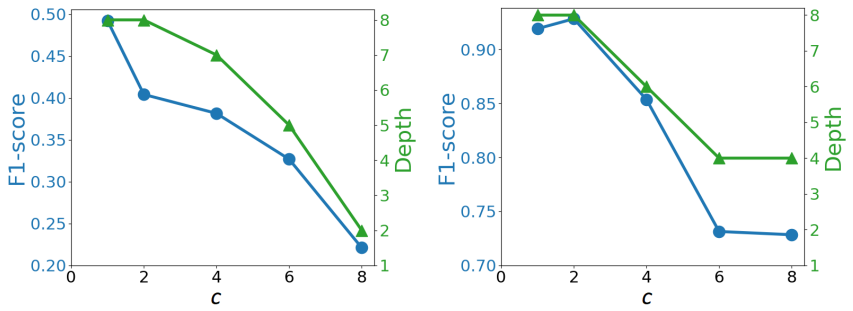
Figure 4.5: Median minimum path cost to a goal node found at a given time, across 3 trials (for trials that terminate first, we extend the plots so the median remains monotonic). A*-NEAR (blue) and IDS-BB-NEAR (green) will often find a goal node with a smaller path cost, or find one of similar performance but much faster.

Cost vs. performance trade-off. We can also consider a modification of our objective in (4.1) that allows us to use a hyperparameter c to control the trade-off between structural cost (a proxy for interpretability) and performance:

$$(\alpha^*, \theta^*) = \arg\min_{(\alpha,\theta)} (c \cdot s(\alpha) + \zeta(\alpha, \theta)). \tag{4.9}$$

To visualize this trade-off, we run A*-NEAR with the modified objective (4.9) for various values of c. Note that $c = 1$ is equivalent to our experiments in Table 4.1. Figure 4.6 shows that for the *Basketball* and *CRIM13* datasets, as we increase c, which puts more weight on the structural cost, the resulting programs found by A*-NEAR search have decreasing F1-scores but are also more shallow. This confirms our expectations that we can control the trade-off between structural cost

and performance, which allows users of NEAR-guided search algorithms to adjust to their preferences. Unlike the other two experimental domains, the best performing programs learned in *Fly-vs.-Fly* were relatively shallow, so we omitted this domain as the trade-off showed little change in program depth.



(a) CRIM13-sniff (b) Bball-ballhandler

Figure 4.6: As we increase c in (4.9), we observe that A*-NEAR will learn programs with decreasing program depth and also decreasing F1-score. This highlights that we can use c to control the trade-off between structural cost and performance.

We illustrate the implications of this trade-off on interpretability using the depth-2 program in Figure 4.7 and the depth-8 program in Figure 4.8, both synthesized for the same task of detecting a "sniff" action in the CRIM13 dataset. The depth-2 program says that a "sniff" occurs if the intruder mouse is close to the right side of the cage and both mice are near the bottom of the cage, and can be seen to apply a *position bias* (regarding the location of the action) on the action. This program is simple, due to the large weight on the structural cost, and has a low F1-score. In contrast, the deeper program in Figure 4.8 has performance comparable to an RNN but is more difficult to interpret. Our interpretation of this program is that it evaluates the likelihood of "sniff" by applying a position bias, then using the velocity of the mice if the mice are close together and not moving fast, and using distance between the mice otherwise.

4.5 Related Work

Neural program induction. The literature on *neural program induction* (NPI) (Graves, Wayne, and Danihelka, 2014; Kurach, Andrychowicz, and Sutskever, 2015; Reed and De Freitas, 2015; Santoro et al., 2016) develops methods to learn neural networks that can perform procedural (program-like) tasks, typically using architectures augmented with differentiable memory. Our approach differs from these methods in that its final output is a symbolic program. However, since our

```
mapprefix(
  SlidingWindowAverage(PositionAffine(s_t)),
τ)
```

Figure 4.7: Synthesized depth-2 program classifying a "sniff" action between two mice in the CRIM13 dataset. The sliding window average is over the last 10 frames. The program achieves F1-score of 0.22 (vs. 0.48 for RNN baseline). This program is synthesized using $c = 8$.

```
map(
  add(
    PositionAffine(s_t),
    if (add(VelocityAffine(s_t), DistAffine(s_t)) > 0)
      then VelocityAffine(s_t)
    else DistAffine(s_t)
  ),
τ)
```

Figure 4.8: Synthesized depth 8-program classifying a "sniff" action between two mice in the CRIM13 dataset. The program achieves F1-score of 0.46 (vs. 0.48 for RNN baseline). This program is synthesized using $c = 1$.

heuristics are neural approximation of programs, our work can be seen as repeatedly performing NPI as the program is being produced. While we have so far used classical feedforward and recurrent architectures to implement our neural heuristics, future work could use richer models from the NPI literature to this end.

DSL-based program synthesis. There is a large body of research on synthesis of programs from DSLs. In many of these methods, the goal is not learning but finding a program that satisfies a hard constraint (Alur et al., 2015; Solar-Lezama et al., 2006; Polozov and Gulwani, 2015; Feser, Chaudhuri, and Dillig, 2015). However, there is also a growing literature on learning programs from (noisy) data (Lake, Salakhutdinov, and Joshua B Tenenbaum, 2015; Ellis, Solar-Lezama, and Joshua B. Tenenbaum, 2015; Verma, Murali, et al., 2018; Ellis, Ritchie, et al., 2018; Valkov et al., 2018; Verma, H. M. Le, et al., 2019). Of these methods, TERPRET (Gaunt, Brockschmidt, Singh, et al., 2016) and NEURAL TERPRET (Gaunt, Brockschmidt, Kushman, et al., 2017) allows gradient descent as a mechanism for learning program parameters. However, unlike NEAR, these approaches do not allow a general search

over program architectures permitted by a DSL, and require a detailed hand-written template of the program for even the simplest tasks. While the Houdini framework (Valkov et al., 2018) combines gradient-based parameter learning with search over program architectures, this search is not learning-accelerated and uses a simple type-directed enumeration. As reported in our experiments, NEAR outperforms this enumeration-based approach.

Many recent methods for program synthesis use statistical models to guide the search over program architectures (Balog et al., 2017; Chen, C. Liu, and Song, 2019; Ellis, Solar-Lezama, and J. Tenenbaum, 2016; Devlin et al., 2017; Ellis, Ritchie, et al., 2018; Parisotto et al., 2016; Ganin et al., 2018; Murali, Chaudhuri, and Jermaine, 2018). In particular, (Lee et al., 2018) use a probabilistic model to guide an A*search over programs. Most of these models (including the one in (Lee et al., 2018)) are trained using corpora of synthesis problems and corresponding solutions, which are not available in our setting. There is a category of methods based on reinforcement learning (RL) (Ganin et al., 2018; Bunel et al., 2018). Unlike NEAR, these methods do not directly exploit the structure of the search space. Combining them with our approach would be an interesting topic of future work.

Structure search using relaxations. Our search problem bears similarities with the problems of searching over neural architectures and the structure of graphical models. Prior work has used relaxations to solve these problems (H. Liu, Simonyan, and Yang, 2019; Shin, Packer, and Song, 2018; Zoph and Q. V. Le, 2017; Real, Huang, and Q. V. Le, 2019; Xiang and Kim, 2013). Specifically, the A* lasso approach for learning sparse Bayesian networks uses a dense network to construct admissible heuristics (Xiang and Kim, 2013), and DARTS computes a differentiable relaxation of neural architecture search (H. Liu, Simonyan, and Yang, 2019; Shin, Packer, and Song, 2018). The key difference between these efforts and ours is that the design space in our problem is much richer, making the methods in prior work difficult to apply. In particular, DARTS uses a composition of softmaxes over all possible candidate operations between a fixed set of nodes that constitute a neural architecture, and the heuristics in the A* lasso method come from a single, simple function class. However, in our setting, there is no fixed bound on the number of expressions in a program, different sets of operations can be available at different points of synthesis, and the input and output type of the heuristic (and therefore, its architecture) can vary based on the part of the program derived so far.

4.6 Discussion

We have a presented a novel graph search approach to learning differentiable programs. Our method leverages a novel construction of an admissible heuristic using neural relaxations to efficiently search over program architectures. Our experiments show that programs learned using our approach can have competitive performance, and that our search-based learning procedure substantially outperforms conventional program learning approaches.

There are many directions for future work. One direction is to extend the approach to richer DSLs and neural heuristic architectures, for example, those suited to reinforcement learning (Verma, H. M. Le, et al., 2019) and generative modeling (Ritchie et al., 2016). Another is to combine NEAR with classical program synthesis methods based on symbolic reasoning. A third is to integrate NEAR into more complex program search problems, e.g., when there is an initial program as a starting point and the goal is to search for refinements. A fourth is to more tightly integrate with real-world applications to evaluate the interpretability of learned programs as it impacts downstream tasks.

References

Alur, Rajeev et al. (2015). "Syntax-Guided Synthesis". In: *Dependable Software Systems Engineering*, pp. 1–25. DOI: 10.3233/978-1-61499-495-4-1.

Bagchi, A and A Mahanti (1983). "Admissible heuristic search in AND/OR graphs". In: *Theoretical Computer Science* 24.2, pp. 207–219.

Balog, Matej et al. (2017). "DeepCoder: Learning to Write Programs". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

Bunel, Rudy et al. (2018). "Leveraging grammar and reinforcement learning for neural program synthesis". In: *arXiv preprint arXiv:1805.04276*.

Burgos-Artizzu, Xavier P et al. (2012). "Social behavior recognition in continuous video". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, pp. 1322–1329.

Charniak, Eugene and Saadia Husain (1991). *A new admissible heuristic for minimal-cost proofs*. Brown University, Department of Computer Science.

Chen, Xinyun, Chang Liu, and Dawn Song (2019). "Execution-Guided Neural Program Synthesis". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: https://openreview.net/forum?id=H1gfOiAqYm.

Devlin, Jacob et al. (2017). "Robustfill: Neural program learning under noisy I/O". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, pp. 990–998.

Dietterich, Thomas G. (2002). "Machine Learning for Sequential Data: A Review". In: *Structural, Syntactic, and Statistical Pattern Recognition, Joint IAPR International Workshops SSPR 2002 and SPR 2002, Windsor, Ontario, Canada, August 6-9, 2002, Proceedings*, pp. 15–30. DOI: 10.1007/3-540-70659-3_2. URL: https://doi.org/10.1007/3-540-70659-3%5C_2.

Ellis, Kevin, Maxwell I. Nye, et al. (2019). "Write, Execute, Assess: Program Synthesis with a REPL". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach et al., pp. 9165–9174. URL: http://papers.nips.cc/paper/9116-write-execute-assess-program-synthesis-with-a-repl.

Ellis, Kevin, Daniel Ritchie, et al. (2018). "Learning to infer graphics programs from hand-drawn images". In: *Advances in Neural Information Processing Systems*, pp. 6059–6068.

Ellis, Kevin, Armando Solar-Lezama, and Josh Tenenbaum (2016). "Sampling for Bayesian Program Learning". In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by Daniel D. Lee et al., pp. 1289–1297. URL: http://papers.nips.cc/paper/6082-sampling-for-bayesian-program-learning.

Ellis, Kevin, Armando Solar-Lezama, and Joshua B. Tenenbaum (2015). "Unsupervised Learning by Program Synthesis". In: *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pp. 973–981. URL: http://papers.nips.cc/paper/5785-unsupervised-learning-by-program-synthesis.

Eyjolfsdottir, Eyrun et al. (2014). "Detecting social actions of fruit flies". In: *European Conference on Computer Vision*. Springer, pp. 772–787.

Feser, John K., Swarat Chaudhuri, and Isil Dillig (2015). "Synthesizing data structure transformations from input-output examples". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pp. 229–239. DOI: 10.1145/2737924.2737977. URL: https://doi.org/10.1145/2737924.2737977.

Ganin, Yaroslav et al. (2018). "Synthesizing Programs for Images using Reinforced Adversarial Learning". In: *CoRR* abs/1804.01118. arXiv: 1804.01118. URL: http://arxiv.org/abs/1804.01118.

Gaunt, Alexander L, Marc Brockschmidt, Nate Kushman, et al. (2017). "Differentiable programs with neural libraries". In: *International Conference on Machine Learning*, pp. 1213–1222.

Gaunt, Alexander L, Marc Brockschmidt, Rishabh Singh, et al. (2016). "Terpret: A probabilistic programming language for program induction". In: *arXiv preprint arXiv:1608.04428*.

Graves, Alex, Greg Wayne, and Ivo Danihelka (2014). "Neural turing machines". In: *arXiv preprint arXiv:1410.5401*.

Harris, Larry R (1974). "The heuristic search under conditions of error". In: *Artificial Intelligence* 5.3, pp. 217–234.

Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman (2007). *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley. ISBN: 978-0-321-47617-3.

Kocsis, Levente and Csaba Szepesvári (2006). "Bandit based monte-carlo planning". In: *European conference on machine learning*. Springer, pp. 282–293.

Korf, Richard E (2000). "Recent progress in the design and analysis of admissible heuristic functions". In: *International Symposium on Abstraction, Reformulation, and Approximation*. Springer, pp. 45–55.

Kurach, Karol, Marcin Andrychowicz, and Ilya Sutskever (2015). "Neural random-access machines". In: *arXiv preprint arXiv:1511.06392*.

Lake, Brenden M, Ruslan Salakhutdinov, and Joshua B Tenenbaum (2015). "Human-level concept learning through probabilistic program induction". In: *Science* 350.6266, pp. 1332–1338.

Lee, Woosuk et al. (2018). "Accelerating search-based program synthesis using learned probabilistic models". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pp. 436–449.

Liu, Hanxiao, Karen Simonyan, and Yiming Yang (2019). "DARTS: Differentiable Architecture Search". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. URL: https://openreview.net/forum?id=S1eYHoC5FX.

Murali, Vijayaraghavan, Swarat Chaudhuri, and Chris Jermaine (2018). "Neural Sketch Learning for Conditional Program Generation". In: *ICLR*.

Parisotto, Emilio et al. (2016). "Neuro-symbolic program synthesis". In: *arXiv preprint arXiv:1611.01855*.

Pearl, Judea (1984). "Heuristics: Intelligent search strategies for computer problem solving". In: *Addision Wesley*.

Polozov, Oleksandr and Sumit Gulwani (2015). "FlashMeta: a framework for inductive program synthesis". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pp. 107–126.

Real, Esteban, Yanping Huang, and Quoc V. Le (2019). "Regularized Evolution for Image Classifier Architecture Search". In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, pp. 4780–4789. DOI: 10.1609/aaai.v33i01.33014780. URL: https://doi.org/10.1609/aaai.v33i01.33014780.

Reed, Scott and Nando De Freitas (2015). "Neural programmer-interpreters". In: *arXiv preprint arXiv:1511.06279*.

Ritchie, Daniel et al. (2016). "Neurally-guided procedural models: Amortized inference for procedural graphics programs using neural networks". In: *Advances in neural information processing systems*, pp. 622–630.

Russell, Stuart and Peter Norvig (2002). "Artificial intelligence: a modern approach". In:

Santoro, Adam et al. (2016). "Meta-Learning with Memory-Augmented Neural Networks". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, pp. 1842–1850. URL: http://proceedings.mlr.press/v48/santoro16.html.

Sasaki, Yutaka (Jan. 2007). "The truth of the F-measure". In: *Teach Tutor Master*.

Shah, Ameesh et al. (2020). "Learning Differentiable Programs with Admissible Neural Heuristics". In: *Advances in Neural Information Processing Systems*. Vol. 33, pp. 4940–4952. URL: https://proceedings.neurips.cc/paper/2020/hash/342285bb2a8cadef22f667eeb6a63732-Abstract.html.

Shin, Richard, Charles Packer, and Dawn Song (2018). "Differentiable Neural Network Architecture Search". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net. URL: https://openreview.net/forum?id=BJ-MRKkwG.

Solar-Lezama, Armando et al. (2006). "Combinatorial sketching for finite programs". In: *ASPLOS*, pp. 404–415.

Sontag, David et al. (2012). "Tightening LP relaxations for MAP using message passing". In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*.

Valenzano, Richard Anthony et al. (2013). "Using alternative suboptimality bounds in heuristic search". In: *Twenty-Third International Conference on Automated Planning and Scheduling*.

Valkov, Lazar et al. (2018). "HOUDINI: Lifelong Learning as Program Synthesis". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pp. 8701–8712. URL: http://papers.nips.cc/paper/8086-houdini-lifelong-learning-as-program-synthesis.

Verma, Abhinav, Hoang Minh Le, et al. (2019). "Imitation-Projected Programmatic Reinforcement Learning". In: *Advances in Neural Information Processing Systems (NeurIPS)*.

Verma, Abhinav, Vijayaraghavan Murali, et al. (2018). "Programmatically Interpretable Reinforcement Learning". In: *International Conference on Machine Learning*, pp. 5052–5061.

Winskel, Glynn (1993). *The formal semantics of programming languages: an introduction*. MIT press.

Xiang, Jing and Seyoung Kim (2013). "A* Lasso for Learning a Sparse Bayesian Network Structure for Continuous Variables". In: *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*. Ed. by Christopher J. C. Burges et al., pp. 2418–2426. URL: http://papers.nips.cc/paper/5174-a-lasso-for-learning-a-sparse-bayesian-network-structure-for-continuous-variables.

Young, Halley, Osbert Bastani, and Mayur Naik (2019). "Learning neurosymbolic generative models via program synthesis". In: *International Conference on Machine Learning (ICML)*.

Yue, Yisong et al. (2014). "Learning fine-grained spatial models for dynamic sports play prediction". In: *ICDM*.

Zoph, Barret and Quoc V. Le (2017). "Neural Architecture Search with Reinforcement Learning". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. URL: https://openreview.net/forum?id=r1Ue8Hcxg.

Chapter 5

LEARNING NEUROSYMBOLIC ENCODERS FOR INTERPRETABLE REPRESENTATIONS

The work in this chapter was published in (Zhan, Sun, et al., 2021). E.Z. participated in the conception of the project, formulated and implemented the method, conducted experiments and analyzed results, and participated in the writing of the manuscript.

Summary

We present a framework for the unsupervised learning of neurosymbolic encoders, i.e., encoders obtained by composing neural networks with symbolic programs from a domain-specific language. Such a framework can naturally incorporate symbolic expert knowledge into the learning process and lead to more interpretable and factorized latent representations than fully neural encoders. Also, models learned this way can have downstream impact, as many analysis workflows can benefit from having clean programmatic descriptions. We ground our learning algorithm in the variational autoencoding (VAE) framework, where we aim to learn a neurosymbolic encoder in conjunction with a standard decoder. Our algorithm integrates standard VAE-style training with modern program synthesis techniques. We evaluate our method on learning latent representations for real-world trajectory data from animal biology and sports analytics. We show that our approach offers significantly better separation than standard VAEs and leads to practical gains on downstream tasks.

5.1 Introduction

Advances in unsupervised learning have enabled the discovery of latent structures in data from a variety of domains, such as image data (Dupont, 2018), sound recordings (Calhoun, Pillow, and Murthy, 2019), and tracking data (Luxem et al., 2020). For instance, a common approach is to use encoder-decoder frameworks, such as variational autoencoders (VAE) (Kingma and Welling, 2014), to identify a low-dimensional latent representation from the raw data that could contain disentangled factors of variation (Dupont, 2018) or semantically meaningful clusters (Luxem et al., 2020). Such approaches typically employ complex mappings based on neural networks, which can make it difficult to explain how the model assigns inputs to latent representations (Y. Zhang et al., 2020).

To address this issue, we introduce *unsupervised neurosymbolic representation learning*, where the goal is to find a programmatically interpretable representation (as part of a larger neurosymbolic representation) of the raw data. We consider programs to be differentiable, symbolic models instantiated using a domain-specific language (DSL), and use neurosymbolic to refer to blendings of neural and symbolic. Neurosymbolic encoders can offer a few key benefits. First, since the DSL reflects structured domain knowledge, they can often be human-interpretable (Verma et al., 2018; Shah et al., 2020). Second, by leveraging the inductive bias of the DSL, neurosymbolic encoders can potentially offer more factorized or well-separated representations of the raw data (i.e., the representations are more semantically meaningful), which has been observed in studies that used hand-crafted programmatic encoders (Zhan, Tseng, et al., 2020).

Our learning algorithm is grounded in the VAE framework (Kingma and Welling, 2014; Mnih and Gregor, 2014), where the goal is to learn a neurosymbolic encoder coupled with a standard neural decoder. A key challenge is that the space of programs is combinatorial, which we tackle via a tight integration between standard VAE training with modern program synthesis methods (Shah et al., 2020). We further show how to incorporate ideas from adversarial information factorization (Creswell et al., 2017) and enforcing capacity constraints (Burgess et al., 2017; Dupont, 2018) in order to mitigate issues such as posterior and index collapse in the learned representation.

We evaluate our neurosymbolic encoding approach on multiple behavior analysis domains, where the data are from challenging real-world settings and cluster interpretability is important for domain experts. Our contributions are:

- We propose a novel unsupervised approach to train neurosymbolic encoders, to result in a programmatically interpretable representation of data (as part of a neurosymbolic representation).
- We show that our approach can significantly outperform purely neural encoders in extracting semantically meaningful representations of behavior, as measured by standard unsupervised metrics.
- We further explore the flexibility of our approach, by showing that performance can be robust across different DSL designs by domain experts.

- We showcase the practicality of our approach in the downstream task of *task programming*, a state-of-the-art self-supervised learning approach for behavior analysis (Sun, Kennedy, et al., 2021).

5.2 Preliminaries: VAEs and Differentiable Program Synthesis

Variational Autoencoders

We build upon VAEs (Kingma and Welling, 2014; Mnih and Gregor, 2014), a latent variable modeling framework shown to learn effective latent representations (also called encodings/embeddings) (Higgins et al., 2016; Zhao, J. Song, and Ermon, 2017; Yingzhen and Mandt, 2018) and can capture the generative process (Oord, Vinyals, and Kavukcuoglu, 2017; Vahdat and Kautz, 2020; Zhan, Tseng, et al., 2020). VAEs introduce a latent variable $\mathbf{z}$, an encoder q_ϕ, a decoder p_θ, and a prior distribution p on $\mathbf{z}$. ϕ and θ are the parameters of the q and p respectively, often instantiated with neural networks. The learning objective is to maximize the evidence lower bound (ELBO) of the data log-likelihood:

$$\text{ELBO} := \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}|\mathbf{z})\right] - D_{KL}\big(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z})\big) \leq \log p(\mathbf{x}). \tag{5.1}$$

The first term in (5.1) is the log-density assigned to the data, while the second term is the KL-divergence between the prior and approximate posterior of $\mathbf{z}$. Latent representations $\mathbf{z}$ are often continuous and modeled with a Gaussian prior, but $\mathbf{z}$ can be modeled to contain discrete dimensions as well (Kingma, Rezende, et al., 2014; Hu et al., 2017; Dupont, 2018). Our experiments are focused on behavioral tracking data in the form of trajectories, and so in practice we utilize a trajectory variant of VAEs (Co-Reyes et al., 2018; Zhan, Tseng, et al., 2020; Sun, Kennedy, et al., 2021), described further at the end of Section 5.3.

One challenge with VAEs (and deep encoder-decoder models in general) is that while the model is expressive, it is often difficult to interpret what is encoded in the latent representation $\mathbf{z}$. Common approaches include taking traversals in the latent space and visualizing the resulting generations (Burgess et al., 2017), or post-processing the latent variables using techniques such as clustering (Luxem et al., 2020). Such techniques are post-hoc and thus cannot guide (in an interpretable way) the encoder to be biased towards a family of structures. Some recent work have studied how to impose structure in the form of graphical models or dynamics in the latent space (Johnson et al., 2016; Deng et al., 2017). Our work can be thought of as a first step towards imposing structure in the form of symbolic knowledge encoded in a domain specific programming language.

Synthesis of Differentiable Programs

Our approach utilizes recent work on the synthesis of differentiable programs (Shah et al., 2020; Valkov et al., 2018), where one learns both the discrete structure of the symbolic program (analogous to the architecture of a neural network) as well as differentiable parameters within that structure. In particular, we use the formulation that we introduced in Chapter 4, which we summarize below.

We use a domain-specific functional programming language (DSL), generated with a context-free grammar (see Figure 5.2 for an example). Programs are represented as a pair (α, ψ), where α is a complete program architecture (no nonterminals) and ψ are its real-valued parameters. We denote $\mathcal{P}$ as the space of symbolic programs (i.e. programs with complete architectures). The semantics of a program (α, ψ) are given by a function $[[\alpha]](x, \psi)$, which is guaranteed by the semantics of the DSL to be differentiable in both x and ψ.

Like in Chapter 4, we pose the problem of learning differentiable programs as search through a directed program graph $\mathcal{G}$. The graph $\mathcal{G}$ models the top-down construction of program architectures through the repeated firing of rules of the DSL grammar, starting with an *empty* architecture (represented by the "start" nonterminal of the grammar). Thus $\mathcal{P}$, the set of complete programs, is simply the set of leaf nodes of $\mathcal{G}$. Other non-leaf nodes in $\mathcal{G}$ contain programs with *partial* architectures (has at least one nonterminal), which we denote with u. We interpret a program in a non-leaf node as being neurosymbolic, by viewing its nonterminals as representing neural networks with free parameters (i.e., completion of a partial architectures). The source node in $\mathcal{G}$ is the empty architecture u_0, interpreted as a fully neural program. A directed edge (u, u') exists in $\mathcal{G}$ if one can obtain u' from u by applying a rule in the DSL that replaces a nonterminal in u. In that case, we would call u' a *child* of u.

Program synthesis in this problem setting equates to searching through $\mathcal{G}$ to find the optimal complete program architecture, and then learning corresponding parameters ψ, i.e., to find the optimal (α, ψ) that minimizes a combination of standard training loss (e.g., classification error) and structural loss (preferring "simpler" α's). In Chapter 4, we evaluated multiple strategies for solving this problem and found *informed search using admissible neural heuristics* to be the most efficient strategy. Consequently, we adopt this same algorithm for our program synthesis task.

5.3 Neurosymbolic Encoders

The structure of our neurosymbolic encoder is shown in the right diagram of Figure 5.1. The latent representation $\mathbf{z} = [\mathbf{z}_\phi, \mathbf{z}_{(\alpha,\psi)}]$ is partitioned into neurally encoded $\mathbf{z}_\phi$ and programmatically encoded $\mathbf{z}_{(\alpha,\psi)}$. This approach boasts several advantages:

- The symbolic component of the latent representation is programmatically interpretable.
- The neural component can encode any residual information not captured by the program, which maintains the model's capacity compared to standard deep encoders.
- By incorporating a modular design, we can leverage state-of-the-art learning algorithms for both differentiable encoder-decoder training and program synthesis.

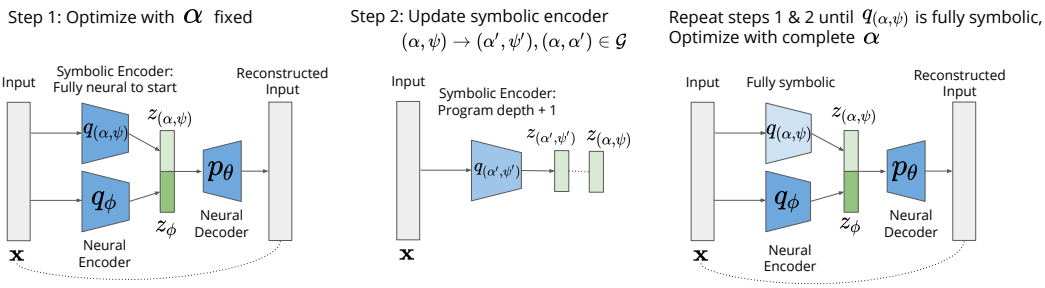


Figure 5.1: Sketch of Algorithm 4. The symbolic encoder is initially fully neural. We alternate between VAE training with the program architecture fixed (Step 1), and supervised program learning to increase the depth of the program by 1 (Step 2). Once we reach a symbolic program, we train the model one last time to learn all the parameters. The color (in terms of lightness) of the symbolic encoder corresponds to the encoder becoming more symbolic over time.

We denote q_ϕ and $q_{(\alpha,\psi)}$ as the neural and symbolic encoders respectively (see Figure 5.1), where $\mathbf{z}_\phi \sim q_\phi(\cdot|\mathbf{x})$ and $\mathbf{z}_{(\alpha,\psi)} \sim q_{(\alpha,\psi)}(\cdot|\mathbf{x})$. q_ϕ is instantiated with a neural network, but $q_{(\alpha,\psi)}$ is a differentiable program with architecture α and parameters ψ in the symbolic program space $\mathcal{P}$ defined by a DSL. Given an unlabeled training

set of $\mathbf{x}$'s, the VAE learning objective in (5.1) then becomes:

$$\begin{aligned}\max_{\phi,(\alpha,\psi),\theta} \quad \mathop{\mathbb{E}}_{q_\phi(\mathbf{z}_\phi|\mathbf{x})q_{(\alpha,\psi)}(\mathbf{z}_{(\alpha,\psi)}|\mathbf{x})} & \Big[\underbrace{\log p_\theta(\mathbf{x}|\mathbf{z}_\phi, \mathbf{z}_{(\alpha,\psi)})}_{\text{reconstruction loss}} \Big] \\ & - \underbrace{D_{KL}\big(q_\phi(\mathbf{z}_\phi|\mathbf{x})||p(\mathbf{z}_\phi)\big)}_{\text{regularization for neural latent}} \\ & - \underbrace{D_{KL}\big(q_{(\alpha,\psi)}(\mathbf{z}_{(\alpha,\psi)}|\mathbf{x})||p(\mathbf{z}_{(\alpha,\psi)})\big)}_{\text{regularization for symbolic latent}}.\end{aligned} \tag{5.2}$$

Compared to the standard VAE objective in (5.1) for a single neural encoder, (5.2) has separate KL-divergence terms for the neural and programmatic encoders.

Learning Algorithm

The challenge with solving for (5.2) is that while (ϕ, ψ, θ) can be optimized via back-propagation with α fixed, optimizing for α is a discrete optimization problem. Since it is difficult to jointly optimize over both continuous and discrete spaces, we take an iterative, alternating optimization approach. We start with a fully neural program (one with empty architecture u_0) trained using standard differentiable optimization (Figure 5.1, Step 1). We then gradually make it more symbolic (Figure 5.1, Step 2) by finding a program that is a child of the current program in $\mathcal{G}$ (more symbolic by construction of $\mathcal{G}$) that outputs as similar to the current latent representations as possible:

$$\min_{u':(u,u')\in\mathcal{G},\ \psi'} \quad \mathcal{L}_{\text{supervised}}\big(q_{(u,\psi)}(\mathbf{x}), q_{(u',\psi')}(\mathbf{x})\big), \tag{5.3}$$

which can be viewed as a form of distillation (from less symbolic to more symbolic programs) via matching the input/output behavior. We solve (5.3) by enumerating over all child programs and selecting the best one, which is similar to iteratively-deepened depth-first search in Chapter 4. We alternate between optimizing (5.2) and (5.3) until we obtain a program with complete architecture α. Algorithm 4 outlines this procedure and is guaranteed to terminate if $\mathcal{G}$ is finite by specifying a maximum program depth.

We chose this optimization procedure for two reasons. First, it maximally leverages state-of-the-art tools in both differentiable latent variable modeling (VAE-style training) and supervised program synthesis, leading to tractable algorithm design. Second, this procedure never makes a drastic change to the program architecture, leading to relatively stable learning behavior across iterations.

Algorithm 4 Learning a neurosymbolic encoder

Input: program graph $\mathcal{G}$, symbolic program space $\mathcal{P}$
Initialize $\phi, \psi, \theta, u = u_0$ (empty architecture)
while u is not complete **do**
    $\phi, \psi, \theta \leftarrow$ optimize (5.2) with neural completion of u
    $(u, \psi) \leftarrow$ optimize (5.3)
$\phi, \psi, \theta \leftarrow$ optimize (5.2) with complete architecture $u = \alpha$
return encoder $\{q_\phi, q_{(\alpha,\psi)}\}$

Algorithm 5 Learning a neurosymbolic encoder with k programs

Input: program graph $\mathcal{G}$, symbolic program space $\mathcal{P}$, k
for $i = 1..k$ **do**
    fix programs $\{q_{(\alpha_1,\psi_1)}, \ldots, q_{(\alpha_{i-1},\psi_{i-1})}\}$
    execute Algorithm 4 to learn $q_{(\alpha_i,\psi_i)}$
    remove $q_{(\alpha_i,\psi_i)}$ from $\mathcal{P}$ to avoid redundancies
return encoder $\{q_\phi, q_{(\alpha_1,\psi_1)}, \ldots, q_{(\alpha_k,\psi_k)}\}$

Learning Multiple Programs

The interpretability of latent representations induced by symbolic encoders $q_{(\alpha,\psi)}$ ultimately depends on the DSL. For instance, a program that encodes to one of ten classes may not be very interpretable if it involves a matrix multiplication within the program. Instead, we learn *binary* programs that encode sequences into one of two classes (using binary cross-entropy for $\mathcal{L}_{\text{supervised}}$, a uniform prior on $\mathbf{z}_{(\alpha,\psi)}$, and Gumbel-Softmax (Jang, Gu, and Poole, 2017) to sample from the posterior). Figures 5.5a & 5.5b depict learned binary programs that encode mice trajectories and their interpretations.

To encode more than two classes, we can simply learn multiple binary programs by extending (5.2) to sum $\mathcal{L}_{\text{supervised}}$ over k symbolic programs $\{q_{(\alpha_1,\psi_1)}, \ldots, q_{(\alpha_k,\psi_k)}\}$ and corresponding latent representations $\{\mathbf{z}_{(\alpha_1,\psi_1)}, \ldots, \mathbf{z}_{(\alpha_k,\psi_k)}\}$. This results in 2^k classes and a solution space that now scales exponentially (e.g. $|\mathcal{P}|^k$ if using exhaustive enumeration). Algorithm 5 outlines our greedy solution that reuses Algorithm 4 by iteratively learning one symbolic program at a time. We leave the exploration of more sophisticated search methods as future work.

Dealing with Posterior and Index Collapse

Deep latent variable models, especially those with discrete latent variables, are notoriously prone to both posterior (Bowman et al., 2015; Xi Chen, Kingma, et

al., 2017; Oord, Vinyals, and Kavukcuoglu, 2017) and index (Kaiser et al., 2018) collapse. Since our algorithms optimize for such models repeatedly, we can be more susceptible to these failure modes. There are many approaches available for tackling both these issues, but we emphasize that these contributions are orthogonal to ours; as techniques for preventing posterior and index collapse improve, so will the robustness of our algorithm. Below, we summarize two strategies that we found to work well in our setting.

Adversarial information factorization (Creswell et al., 2017) introduces an adversarial network A_ω that aims to predict $\mathbf{z}_{(\alpha,\psi)}$ from $\mathbf{z}_\phi$. Maximizing this adversarial loss can prevent index collapse, where all data is encoded into the same class, as doing so would would fail to fool the adversary.

$$\begin{aligned}\max_{\phi,(\alpha,\psi),\theta} \quad \mathop{\mathbb{E}}_{q_\phi(\mathbf{z}_\phi|\mathbf{x})q_{(\alpha,\psi)}(\mathbf{z}_{(\alpha,\psi)}|\mathbf{x})} \Big[\log p_\theta(\mathbf{x}|\mathbf{z}_\phi, \mathbf{z}_{(\alpha,\psi)}) + \underbrace{\min_\omega \mathcal{L}_{\text{adv}}\big(A_\omega(\mathbf{z}_\phi), \mathbf{z}_{(\alpha,\psi)}\big)}_{\text{adversary}} \Big] \\ - D_{KL}\big(q_\phi(\mathbf{z}_\phi|\mathbf{x})||p(\mathbf{z}_\phi)\big) \\ - D_{KL}\big(q_{(\alpha,\psi)}(\mathbf{z}_{(\alpha,\psi)}|\mathbf{x})||p(\mathbf{z}_{(\alpha,\psi)})\big)\end{aligned} \tag{5.4}$$

Channel capacity constraint (Burgess et al., 2017; Dupont, 2018) forces the KL-divergence terms to match capacities C_ϕ and $C_{(\alpha,\psi)}$. Since the KL-divergence is an upper bound on the mutual information between latent variables and the data (Kim and Mnih, 2018; Dupont, 2018), this encourages the latent variables to encode information and aims to prevent posterior collapse.

$$\begin{aligned}\max_{\phi,(\alpha,\psi),\theta} \quad \mathop{\mathbb{E}}_{q_\phi(\mathbf{z}_\phi|\mathbf{x})q_{(\alpha,\psi)}(\mathbf{z}_{(\alpha,\psi)}|\mathbf{x})} \Big[\log p_\theta(\mathbf{x}|\mathbf{z}_\phi, \mathbf{z}_{(\alpha,\psi)}) \Big] \\ - \gamma_\phi |D_{KL}\big(q_\phi(\mathbf{z}_\phi|\mathbf{x})||p(\mathbf{z}_\phi)\big) - C_\phi| \\ - \gamma_{(\alpha,\psi)} |D_{KL}\big(q_{(\alpha,\psi)}(\mathbf{z}_{(\alpha,\psi)}|\mathbf{x})||p(\mathbf{z}_{(\alpha,\psi)})\big) - C_{(\alpha,\psi)}|\end{aligned} \tag{5.5}$$

In our algorithms, we augment our initial objective in (5.2) with (5.4) and (5.5).

Instantiation for Sequential Domains

The objective in (5.2) describes a general problem that is applicable to any domain. In our experiments, we focus on behavior modeling domains with trajectory data. Trajectory data is often used in scientific applications where interpretability is desirable, such as behavior discovery (Luxem et al., 2020; Hsu and Yttri, 2020). The ability to easily explain the learned latent representation using programs can help

domain experts better understand the structure in their data. Additionally, trajectory data is often low dimensional for each timestamp, which helps experts encode domain knowledge into the DSL more easily (Shah et al., 2020; Sun, Kennedy, et al., 2021; Zhan, Tseng, et al., 2020).

Let τ denote a trajectory of length T: $\tau = \{\mathbf{s}_1, \ldots, \mathbf{s}_T\}$, where $\mathbf{s}_t$ denotes the state at time t. The VAE decoder p_θ is essentially a parameterized policy π_θ , although we focus entirely on the encoder in this work. We then factorize the log-density in (5.2) as a product of conditional probabilities:

$$\log p_\theta(\mathbf{x}|\mathbf{z}_\phi, \mathbf{z}_{(\alpha,\psi)}) = \log \pi_\theta(\tau|\mathbf{z}_\phi, \mathbf{z}_{(\alpha,\psi)}) = \sum_{t=1}^{T} \log \pi_\theta(\mathbf{s}_t|\mathbf{s}_{<t}, \mathbf{z}_\phi, \mathbf{z}_{(\alpha,\psi)}). \quad (5.6)$$

When q_ϕ and π_θ are instantiated with recurrent neural networks (RNN), the model is more commonly known as a trajectory-VAE (TVAE) (Co-Reyes et al., 2018; Zhan, Tseng, et al., 2020; Sun, Kennedy, et al., 2021).

As symbolic encoder $q_{(\alpha,\psi)}$ maps sequences to vectors, we adopt a DSL (Figure 5.2) similar to the one that we previously used for sequence classification in Chapter 4 (Figure 4.1). Our DSL is purely functional and contains both basic algebraic operations and parameterized library functions. Domain experts can easily augment the DSL with their own functions, such as selection functions that select subsets of features that they deem potentially important. We ensure that all programs in our DSL are differentiable, utilizing a smooth approximation of the non-differentiable **if**-**then**-**else** construct. Figures 5.5a and 5.5b depict example programs in our DSL.

$$\begin{aligned} \alpha \quad ::= \quad & x \mid \oplus(\alpha_1, \ldots, \alpha_k) \mid \oplus_\theta(\alpha_1, \ldots, \alpha_k) \\ & \textbf{if } \alpha_1 \textbf{ then } \alpha_2 \textbf{ else } \alpha_3 \mid \textbf{sel}_S\ x \mid \textbf{mapaverage}(\alpha_1,\ x) \end{aligned}$$

Figure 5.2: Our DSL for sequential domains in this chapter, similar to the one used Chapter 4 (Figure 4.1). x, $\oplus$, and $\oplus_\theta$ represent inputs, basic algebraic operations, and parameterized library functions, respectively. $\textbf{sel}_S$ selects a subset S of the dimensions of the input x. $\textbf{mapaverage}(e, x)$ applies the function e to every element of a sequence x and returns the average of the results. We employ a differential approximation of the **if**-**then**-**else** construct.

5.4 Experiments

We study our proposed approach on sequential trajectory data from a synthetic dataset to first provide intuition for our algorithm, and then on real-world datasets in neuroscience and sports analytics.

Experiments with Synthetic Dataset

We generate a synthetic dataset of trajectories and run our algorithm to demonstrate the following:

- Programs can capture factors of variation in the data (in our case, 2 discrete factors).
- Information pertaining to captured factors of variation are extracted out of the latent space.

We generate synthetic trajectories by sampling initial positions and velocities from a Gaussian distribution and introducing 2 ground-truth factors of variation as large external forces in the positive/negative x/y directions that affect velocity, totaling to 4 discrete classes. Velocities are fixed for the entire trajectory, but we also sample small Gaussian noise at each timestep. We generate 10k/2k/2k trajectories of length 25 for train/validation/test. Figure 5.3 shows 50 trajectories from the training set. Full details of the synthetic dataset are included in the Appendix D.3.

We visualize the neural latent space in 2 dimensions of a TVAE with 0, 1, and 2 learned programs in Figure 5.4bcd. The initial TVAE latent space contains 4 clusters corresponding to the 4 ground-truth classes in Figure 5.4b. After our algorithm learns the first program that thresholds the final x-position, the resulting latent space in Figure 5.4c captures the other factor of variation as 2 clusters corresponding to the final y-positions. Lastly, when our algorithm learns a second program that thresholds the final y-position, the resulting latent space in Figure 5.4d no longer contains any clear clustering, as we've successfully extracted the 4 ground-truth classes with our programs. Figure 5.4a depicts the 2 learned programs.

Experiments with Real-World Datasets

CalMS21. Our primary real-world dataset is the CalMS21 dataset (Sun, Karigo, et al., 2021), containing trajectories of socially interacting mice captured for neuroscience experiments. Each frame contains 7 tracked keypoints for both mice. The dataset has one set of unlabeled tracking data, which we use to train our neurosymbolic encoder, and another set annotated for 4 behaviors, which we use to evaluate our programs. Specifically, our evaluation uses labels from the test split of the CalMS21 classification task. We have 231k/52k/262k trajectories of length 21 for train/val/test. The features in our DSL are selected by a domain expert based on the attributes from (Segalin et al., 2020).

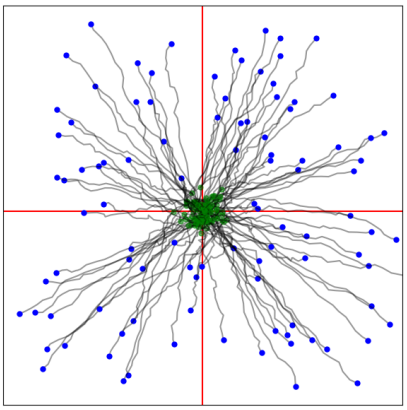
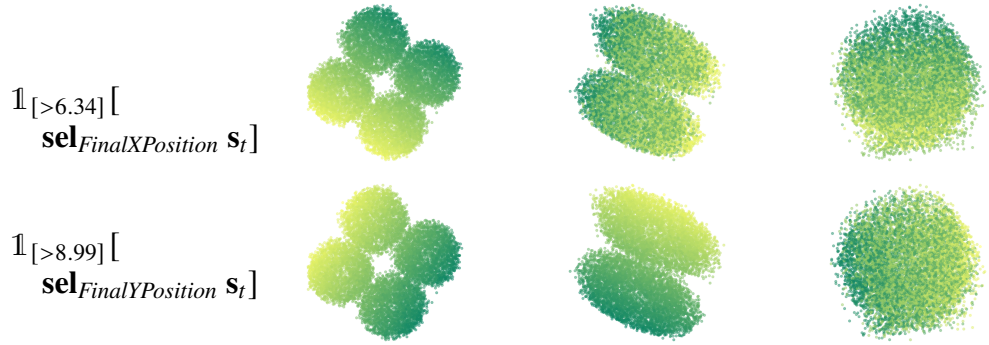


Figure 5.3: Trajectories in synthetic training set. Initial/final positions are indicated in green/blue. Red lines delineate ground-truth classes, based on final positions.



(a) learned programs (b) $\mathbf{z}_\phi$, 0 programs (c) $\mathbf{z}_\phi$, 1 program (d) $\mathbf{z}_\phi$, 2 programs

Figure 5.4: **(a)** $k = 2$ learned binary programs using our algorithm. The first program (top) thresholds the final x-position while the second program (bottom) thresholds the final y-position. **(b, c, d)** Neural latent variables reduced to 2 dimensions. Top/bottom rows are colored by final x/y-positions respectively (green/yellow is positive/negative). **(b)** Clusters in the TVAE neural latent space correspond to 4 ground-truth classes. **(c)** After learning the first program, the neural latent space contains clusters only corresponding on the final y-position. **(d)** After learning the second program, all 4 ground-truth classes have been extracted as programs and the remaining neural latent space contains no clear clustering.

Basketball. We use the same basketball dataset as in previous chapters that tracks professional basketball players. Each trajectory is of length 25 over 8 seconds and contains the xy-positions of 10 players. We split trajectories into two by grouping offensive and defensive players (5 each), effectively doubling the dataset size. We evaluate our algorithm and the baselines with respect to the labels of offensive/defensive players. Our DSL includes additional domain features like player speed and distance-to-basket. In total, we have 177k/31k/27k trajectories for

train/val/test.

Quantitative Evaluation Setup

Baselines. We compare our model containing a neurosymbolic encoder against other approaches based on VAEs and its variations. In particular, we compare against VAE, VAE with k-means loss used in (Ma et al., 2019; Luxem et al., 2020), and Beta-VAE (Burgess et al., 2017). These models have a fully neural encoder and learn continuous latent representations, which we can then use to produce clusters with k-means clustering (Lloyd, 1982). Additionally, we compare against models which produce discrete latent clusterings, such as JointVAEs (Dupont, 2018) and VQ-VAEs (Oord, Vinyals, and Kavukcuoglu, 2017). We use the TVAE version of all baselines (details included in Appendix D.2).

Metrics for evaluation. Unlike in the synthetic setting, we do not have ground truth programs in the real-world datasets. We thus evaluate our programs quantitatively using standard cluster metrics relative to human-defined labels. In particular, we use Purity (Schütze, Manning, and Raghavan, 2008), Normalized Mutual Information (NMI) (H. Zhang et al., 2006), and Rand Index (RI) (Rand, 1971). These metrics have also been used by other works for evaluating clustering such as (Ma et al., 2019; Luxem et al., 2020).

Purity is defined as:

$$Purity = \frac{1}{n} \sum_{u \in U} \max_{v \in V} |u \cap v|, \tag{5.7}$$

where U is the set of human-defined labels, V is the set of cluster assignments from the algorithm, and n is the total number of trajectories.

NMI is defined as:

$$NMI = \frac{\sum_{u \in U} \sum_{v \in V} |u \cap v| \log \left(\frac{n|u \cap v|}{|u||v|} \right)}{\sqrt{\sum_{u \in U} |u| \log \frac{|u|}{n} \sum_{v \in U} |v| \log \frac{|v|}{n}}}. \tag{5.8}$$

RI is defined as:

$$RI = \frac{TP + TN}{n(n-1)/2}, \tag{5.9}$$

where TP are the number of trajectory pairs correctly placed into the same cluster, TN are the number of trajectory pairs correctly placed into different clusters, and n is the total number of trajectories.

For all metrics, a value closer to 1 indicates clusters that more closely match the human-defined labels. We report the median metrics of three trials, and also include a random baseline that assigns a class randomly to each sequence. We include additional results, such as the standard deviation of cluster metrics and the ELBO, in Appendix D.1.

Research Questions

Q1: Are the clusters created with our programs meaningful? We compare clusters produced by our neurosymbolic encoder with fully neural autoencoding baselines (Table 5.1), measured against human-annotated behaviors. For CalMS21, we observe that our method consistently outperforms the baselines in all three cluster metrics. Our method is able to do this by leveraging a DSL which encodes domain knowledge such as behavior attributes that are useful for identifying behaviors. We note that the purity increases as the number of programs (thus clusters) increase, while NMI and RI decreases. This implies our method with two clusters best correspond to CalMS21 behaviors, but the other clusters found by our method may still be useful for domain experts. For Basketball, our method improves slightly with respect to purity, but is overall comparable with the baselines.

We further study the programs and clusters produced by our algorithm for the CalMS21 dataset, through a qualitative study with a domain expert in behavioral neuroscience. In the single program case, the domain expert classified the discovered clusters as when the mice are interacting and when there are no interaction. They noted that this is based on distance between the mice, which is consistent with our program (Figure 5.5a) using distance between nose of resident and tail of intruder. For two programs, there are a total of four clusters, with two clusters each corresponding to no interaction and interaction. For the interaction clusters, the domain expert was further able to identify sniff tail behavior as one of the clusters. In this case, the programs found were based on intruder head body angle, resident nose and intruder tail distance, and resident nose and intruder nose distance. The domain expert found the three program case to be more difficult to interpret, but was able to identify clusters corresponding to sniff tail, resident exploration, interaction facing the same direction (ex: mounting), and interaction facing opposite directions (ex: face-to-face sniffing).

Q2: How sensitive is our approach to different DSL choices? To study the effect of domain expert variation on our approach, we worked with different domain

Model	CalMS21			Basketball		
	Purity	NMI	RI	Purity	NMI	RI
Random assignment	.597	.000	.536	.500	.000	.500
TVAE	.598	.089	.564	.501	.001	.500
TVAE+KMeans loss	.605	.118	.573	.501	.001	.500
JointVAE	.597	.019	.537	.560	.034	.507
VQ-TVAE	.601	.124	.588	.572	.016	.511
Beta-TVAE	.616	.115	.589	.565	.013	.509
Ours (1 program)	.706	.423	.694	.596	.027	.518
Ours (2 programs)	.725	.320	.648	.561	.033	.507
Ours (3 programs)	.756	.314	.633	.584	.022	.514

Table 5.1: Median purity, NMI, and RI on CalMS21 and Basketball compared to human-annotated labels (3 trials). Standard deviations are included in Appendix D.1, and experiment hyperparameters are included in the Appendix D.2.

experts to construct alternate DSLs for studying mouse social behavior on CalMS21. While there is some variation in median cluster metrics, our approach consistently outperforms other baseline approaches that contain fully neural encoders for all three DSLs (Table 5.2). Comparing some learned programs from two DSLs (Figures 5.5a & 5.5b), both contain a term that is correlated with whether the mice is interacting (distance and bounding box overlap), and another term on resident speed (mouse tends to be more stretched when they are moving quickly). A full list of features selected by domain experts is in Appendix D.3.

Model	CalMS21 (DSL 1)			CalMS21 (DSL 2)			CalMS21 (DSL 3)		
	Purity	NMI	RI	Purity	NMI	RI	Purity	NMI	RI
Ours (1 program)	.706	.423	.694	.689	.364	.681	.649	.325	.616
Ours (2 programs)	.725	.320	.648	.715	.359	.673	.664	.324	.634

Table 5.2: Median purity, NMI, and RI on CalMS21 of our algorithms with DSLs selected by three domain experts compared to human-annotated labels (3 runs). DSL1 corresponds to Table 5.1.

Q3: What if we simply encode DSL features as part of trajectory states? Since our method uses features from domain experts as part of our DSL, we additionally experiment with concatenating the same features with trajectory states for training baseline models. For both CalMS21 and Basketball, including features with trajectory states does not lead to any discern able improvements for baseline models (Table 5.3). In contrast, by using the features more explicitly as part of the DSL in

$$\mathbb{1}_{[>-7.02]}\left[\begin{array}{l}\textbf{mapaverage}(\\ \quad\textbf{multiply}(\\ \quad\quad \texttt{ResidentSpeedAffine}_{[-6.28];-8.28}(\mathbf{s}_t),\\ \quad\quad \texttt{NoseTailDistAffine}_{[.042];-9.06}(\mathbf{s}_t)\\ \quad),\\ \tau)\end{array}\right]$$

(a) Program learned using CalMS21 DSL 1, resulting NMI 0.428. Since speed is positive, the first term is always negative. One cluster thus generally consists of trajectories where the mice are further apart, such that the second term is positive, and the negative product is less than the threshold. The other cluster generally occurs when the mice are close together, the second term is negative, and the product will be positive.

$$\mathbb{1}_{[>-5.68]}\left[\begin{array}{l}\textbf{mapaverage}(\\ \quad\textbf{add}(\\ \quad\quad \texttt{ResidentAxisRatioAffine}_{[-7.95];-7.14}(\mathbf{s}_t),\\ \quad\quad \texttt{BoundingBoxIOUAffine}_{[-16.55];5.87}(\mathbf{s}_t)\\ \quad),\\ \tau)\end{array}\right]$$

(b) Program learned using CalMS21 DSL 2, resulting NMI 0.320. The axis ratio is the ratio of major axis length and minor axis length of an ellipse fitted to the mouse keypoints. The second term measures the bounding box overlap between mice, and is zero when the mice are far apart. It follows that one cluster generally contains trajectories when the mice has larger bounding box overlaps or if the resident axis ratio is large. The other cluster thus contains trajectories where the mice bounding boxes do not overlap, and resident body is compact.

Figure 5.5: Learned programs on CalMS21. The subscripts represents the learned weights and biases, in particular, the brackets contain the weights for the affine transformation followed by the bias.

our neurosymbolic encoders, we are able to produce clusters with a better separation between behavior classes based on cluster metrics (as was seen in Table 5.1).

Q4: Are the programs useful for downstream tasks? We integrate the learned programs from our neurosymbolic encoder into the task programming framework (Sun, Kennedy, et al., 2021), a state-of-the-art self- and programmatically-supervised learning approach. This framework uses expert provided programs to train a trajectory representation, which can be applied to behavior analysis. Here, rather than using hand-crafted programs, we instead use the learned programs from our approach, so that experts would only need to provide the DSL. We evaluate on the same mouse dataset (Segalin et al., 2020) as task programming. Using only one program found using our approach, we are able to achieve comparable performance

Model	CalMS21			Basketball		
	Purity	NMI	RI	Purity	NMI	RI
TVAE	.598	.089	.564	.501	.001	.500
TVAE (w/ features)	.597	.103	.570	.565	.012	.508
VQ-TVAE	.601	.124	.588	.571	.016	.511
VQ-TVAE (w/ features)	.608	.114	.601	.525	.002	.501
Beta-TVAE	.616	.115	.589	.566	.013	.509
Beta-TVAE (w/ features)	.612	.096	.571	.563	.011	.508

Table 5.3: Median purity, NMI, and RI on CalMS21 and Basketball compared to human-annotated labels (3 trials) for baselines with trajectories only vs. baselines with trajectories concatenated with DSL features.

to 10 expert-written programs on the behavior classification task studied in (Sun, Kennedy, et al., 2021) (Figure 5.6). This demonstrates that programs found by our approach can be applied effectively to downstream behavior analysis tasks such as task programming.

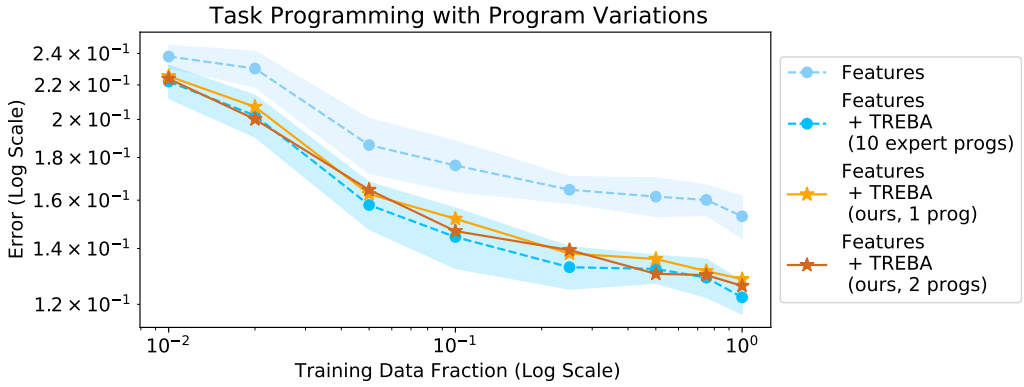


Figure 5.6: Applying symbolic encoders for self-supervision. "Features" is baseline w/o self-supervision. "TREBA" is a self-supervised approach, using either expert-crafted programs or our symbolic encoders as the weak-supervision rules. The shaded region is one standard deviation over 9 repeats. The standard deviation for our approach (not shown) is comparable.

5.5 Related Work

Interpretable latent variable models. Latent representations, especially those that are human-interpretable, can help us understand the structure of data. These models may learn disentangled factors (Higgins et al., 2016; Xi Chen, Duan, et al., 2016) or semantically meaningful clusters (Ma et al., 2019) using unsupervised learning approaches. These approaches are often grounded in the VAE framework

(Kingma and Welling, 2014). Similar to our programs which produce a discrete code, there are other VAE variations which also learns a discrete latent representation, such as JointVAE (Dupont, 2018), Discrete VAE (Rolfe, 2016), and VQ-VAE (Oord, Vinyals, and Kavukcuoglu, 2017). In particular, JointVAE models also learns a discrete and continuous representation. However, these approaches use fully neural encoders. To the best of our knowledge, our work is the first to propose neurosymbolic encoders, where the symbolic component produces an interpretable program.

Neurosymbolic/differentiable program synthesis. Existing program synthesis approaches are often trained in a supervised fashion (Gulwani, 2011; Wang, Dillig, and R. Singh, 2017; Shah et al., 2020), or within a (generative) policy learning context with an explicit reward function (Xinyun Chen, Liu, and D. Song, 2018; Verma et al., 2018; Inala et al., 2020; Feinman and Lake, 2020). In terms of unsupervised program synthesis, the closest related area is generative modeling, as the goal there is to discover programs that can generate the training data (Ellis et al., 2018; Feinman and Lake, 2020), which can be viewed as analogous to learning a symbolic decoder rather than a symbolic encoder. Studying how to incorporate such methods into our framework can be an interesting future direction.

Representation learning for behavior analysis. Representation learning has been applied to a variety of downstream tasks for behavior analysis, such as discovering behavior motifs (Berman et al., 2014; S. H. Singh et al., 2021), identifying internal states (Calhoun, Pillow, and Murthy, 2019), and improving sample-efficiency (Sun, Kennedy, et al., 2021). Studies in this area have used methods such as VAE (Kingma and Welling, 2014), AR-HMM (Wiltschko et al., 2015), and Umap (McInnes, Healy, and Melville, 2018) to better understand the latent structure of behavior. Similar to a few other representation learning methods (Luxem et al., 2020; Sun, Kennedy, et al., 2021), we also use an encoder-decoder setup on trajectory data. However, our work learns a neurosymbolic encoder whereas existing works in this area have fully neural encoders. Our work can aid behavior analysis by learning more interpretable latent representations and can be applied to existing frameworks, such as task programming.

5.6 Discussion

We present a novel approach for unsupervised learning of neurosymbolic encoders. Our approach integrates the VAE framework with program synthesis and results in a learned representation with both neural and symbolic components. Experiments on trajectory data from behavior analysis demonstrate that our programmatic descriptions of the latent space result in more meaningful clusters relative to human-defined behaviors, compared to purely neural encoders. Additionally, we show the practicality of our approach by applying our learned programs to achieve comparable performance to expert-constructed tasks in task programming (Sun, Kennedy, et al., 2021), a self-supervised learning approach for behavior classification.

Based on our work, there are many future directions to explore for neurosymbolic encoders. One direction is based on the observation that interpreting multiple programs simultaneously can still be difficult for domain experts. Combining our approach with other clustering methods, such as hierarchical clustering (Rokach and Maimon, 2005), or working with domain experts to detail more expressive DSLs can help with interpretability. Another direction is to extend this work to other domains such as image and text data, to study both discrete and continuous symbolic latent representations in a more naturally interpretable way. A third direction is to improve upon our greedy approach in Algorithm 5 for finding the optimal set of symbolic programs, e.g. by performing local coordinate ascent in program space, similar to algorithms for large-scale neighborhood search (Ahuja et al., 2002). Lastly, while practically-oriented extensions of VAEs such as our own have yielded great practical benefit, they often lead to sub-optimal results from a pure likelihood (or ELBO) perspective. One final direction is to rigorously formulate a learning objective from the ground up that formally encapsulates practically-oriented extensions of VAEs.

References

Ahuja, Ravindra K et al. (2002). "A survey of very large-scale neighborhood search techniques". In: *Discrete Applied Mathematics* 123.1-3, pp. 75–102.

Berman, Gordon J et al. (2014). "Mapping the stereotyped behaviour of freely moving fruit flies". In: *Journal of The Royal Society Interface* 11.99, p. 20140672.

Bowman, Samuel R. et al. (2015). "Generating Sentences from a Continuous Space". In: *arXiv preprint arXiv:1511.06349*.

Burgess, Christopher P. et al. (2017). "Understanding disentangling in β-VAE". In: *Neural Information Processing Systems Disentanglement Workshop*.

Calhoun, Adam J, Jonathan W Pillow, and Mala Murthy (2019). "Unsupervised identification of the internal states that shape natural behavior". In: *Nature neuroscience* 22.12, pp. 2040–2049.

Chen, Xi, Yan Duan, et al. (2016). "Infogan: Interpretable representation learning by information maximizing generative adversarial nets". In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pp. 2180–2188.

Chen, Xi, Diederik P Kingma, et al. (2017). "Variational lossy autoencoder". In: *International Conference on Learning Representations*.

Chen, Xinyun, Chang Liu, and Dawn Song (2018). "Towards synthesizing complex programs from input-output examples". In: *International Conference on Learning Representations*.

Creswell, Antonia et al. (2017). "Adversarial information factorization". In: *arXiv preprint arXiv:1711.05175*.

Deng, Zhiwei et al. (2017). "Factorized variational autoencoders for modeling audience reactions to movies". In: *IEEE conference on computer vision and pattern recognition*.

Dupont, Emilien (2018). "Learning disentangled joint continuous and discrete representations". In: *Proceedings of the 32nd Conference on Neural Information Processing Systems*.

Ellis, Kevin et al. (2018). "Learning to infer graphics programs from hand-drawn images". In: *Advances in Neural Information Processing Systems*.

Feinman, Reuben and Brenden M Lake (2020). "Learning Task-General Representations with Generative Neuro-Symbolic Modeling". In: *International Conference on Learning Representations*.

Gulwani, Sumit (2011). "Automating string processing in spreadsheets using input-output examples". In: *ACM Sigplan Notices* 46.1, pp. 317–330.

Higgins, Irina et al. (2016). "beta-vae: Learning basic visual concepts with a constrained variational framework". In: *International Conference on Learning Representations*.

Hsu, Alexander I and Eric A Yttri (2020). "B-SOiD: An Open Source Unsupervised Algorithm for Discovery of Spontaneous Behaviors". In: *bioRxiv*, p. 770271.

Hu, Zhiting et al. (2017). "Toward controlled generation of text". In: *International Conference on Machine Learning*.

Inala, Jeevana Priya et al. (2020). "Synthesizing programmatic policies that inductively generalize". In: *International Conference on Learning Representations*.

Jang, Eric, Shixiang Gu, and Ben Poole (2017). "Categorical Reparameterization with Gumbel-Softmax". In: *arXiv preprint arXiv:1611.01144*.

Johnson, Matthew J et al. (2016). “Composing graphical models with neural networks for structured representations and fast inference”. In: *Advances in Neural Information Processing Systems*.

Kaiser, Lukasz et al. (2018). “Fast decoding in sequence models using discrete latent variables”. In: *International Conference on Machine Learning*. PMLR, pp. 2390–2399.

Kim, Hyunjik and Andriy Mnih (2018). “Disentangling by factorising”. In: *International Conference on Machine Learning*.

Kingma, Diederik P, Danilo J Rezende, et al. (2014). “Semi-supervised learning with deep generative models”. In: *arXiv preprint arXiv:1406.5298*.

Kingma, Diederik P and Max Welling (2014). “Auto-encoding variational bayes”. In: *International Conference on Learning Representations*.

Lloyd, Stuart (1982). “Least squares quantization in PCM”. In: *IEEE transactions on information theory* 28.2, pp. 129–137.

Luxem, Kevin et al. (2020). “Identifying behavioral structure from deep variational embeddings of animal motion”. In: *BioRxiv*.

Ma, Qianli et al. (2019). “Learning representations for time series clustering”. In: *Advances in neural information processing systems* 32, pp. 3781–3791.

McInnes, Leland, John Healy, and James Melville (2018). “Umap: Uniform manifold approximation and projection for dimension reduction”. In: *arXiv preprint arXiv:1802.03426*.

Mnih, Andriy and Karol Gregor (2014). “Neural variational inference and learning in belief networks”. In: *International Conference on Machine Learning*.

Oord, Aaron van den, Oriol Vinyals, and Koray Kavukcuoglu (2017). “Neural discrete representation learning”. In: *Proceedings of the 31st Conference on Neural Information Processing Systems*.

Rand, William M (1971). “Objective criteria for the evaluation of clustering methods”. In: *Journal of the American Statistical association* 66.336, pp. 846–850.

Co-Reyes, John D et al. (2018). “Self-consistent trajectory autoencoder: Hierarchical reinforcement learning with trajectory embeddings”. In: *International Conference on Machine Learning (ICML)*.

Rokach, Lior and Oded Maimon (2005). “Clustering methods”. In: *Data mining and knowledge discovery handbook*. Springer, pp. 321–352.

Rolfe, Jason Tyler (2016). “Discrete variational autoencoders”. In: *arXiv preprint arXiv:1609.02200*.

Schütze, Hinrich, Christopher D Manning, and Prabhakar Raghavan (2008). *Introduction to information retrieval*. Vol. 39. Cambridge University Press Cambridge.

Segalin, Cristina et al. (2020). "The Mouse Action Recognition System (MARS): a software pipeline for automated analysis of social behaviors in mice". In: *BioRxiv*.

Shah, Ameesh et al. (2020). "Learning Differentiable Programs with Admissible Neural Heuristics". In: *Advances in Neural Information Processing Systems*. Vol. 33, pp. 4940–4952. URL: https://proceedings.neurips.cc/paper/2020/hash/342285bb2a8cadef22f667eeb6a63732-Abstract.html.

Singh, Satpreet H et al. (2021). "Mining naturalistic human behaviors in long-term video and neural recordings". In: *Journal of Neuroscience Methods*.

Sun, Jennifer J, Tomomi Karigo, et al. (2021). "The Multi-Agent Behavior Dataset: Mouse Dyadic Social Interactions". In: *arXiv preprint arXiv:2104.02710*.

Sun, Jennifer J, Ann Kennedy, et al. (2021). "Task programming: Learning data efficient behavior representations". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2876–2885. URL: https://openaccess.thecvf.com/content/CVPR2021/html/Sun_Task_Programming_Learning_Data_Efficient_Behavior_Representations_CVPR_2021_paper.html.

Vahdat, Arash and Jan Kautz (2020). "Nvae: A deep hierarchical variational autoencoder". In: *Advances in Neural Information Processing Systems*.

Valkov, Lazar et al. (2018). "Houdini: Lifelong learning as program synthesis". In: *Proceedings of the 32nd Conference on Neural Information Processing Systems*.

Verma, Abhinav et al. (2018). "Programmatically interpretable reinforcement learning". In: *International Conference on Machine Learning*. PMLR, pp. 5045–5054.

Wang, Xinyu, Isil Dillig, and Rishabh Singh (2017). "Program synthesis using abstraction refinement". In: *Proceedings of the ACM on Programming Languages*.

Wiltschko, Alexander B et al. (2015). "Mapping sub-second structure in mouse behavior". In: *Neuron* 88.6, pp. 1121–1135.

Yingzhen, Li and Stephan Mandt (2018). "Disentangled sequential autoencoder". In: *International Conference on Machine Learning*.

Zhan, Eric, Jennifer J Sun, et al. (2021). "Unsupervised Learning of Neurosymbolic Encoders". In: *arXiv preprint arXiv:2107.13132*. URL: https://arxiv.org/abs/2107.13132.

Zhan, Eric, Albert Tseng, et al. (2020). "Learning Calibratable Policies using Programmatic Style-Consistency". In: *International Conference on Machine Learning*. PMLR, pp. 11001–11011. URL: https://proceedings.mlr.press/v119/zhan20a.html.

Zhang, Hui et al. (2006). "Unsupervised feature extraction for time series clustering using orthogonal wavelet transform". In: *Informatica* 30.3.

Zhang, Yu et al. (2020). "A Survey on Neural Network Interpretability". In: *arXiv preprint arXiv:2012.14261*.

Zhao, Shengjia, Jiaming Song, and Stefano Ermon (2017). “Learning hierarchical features from generative models”. In: *International Conference on Machine Learning*.

Chapter 6

CONCLUDING REMARKS & FUTURE DIRECTIONS

In this thesis, we have demonstrated the benefits of incorporating programmatic structure with deep learning for behavior modeling. In Chapters 2 and 3, we used programs written by domain experts as sources of weak labels to facilitate learning in challenging applications, namely multi-agent imitation learning and controllable generation of diverse behavior styles. In Chapter 4, we developed NEAR, an efficient algorithm for learning differentiable programs, which we then leverage in Chapter 5 to learn neurosymbolic encoders that automatically learn programmatically-interpretable representations of data. While we consider our work to be among the first steps in programmatic deep learning, there remains many interesting directions yet to be explored, which we outline below.

Putting programmatic deep learning into practice. In our work, an expert-written program or a DSL serves as a starting point for our algorithms, but in practice, we expect a more interactive process with the domain experts. For example, domain experts may continually add library functions to the DSL; can we make sure that NEAR is still efficient as the DSL grows? Domain experts may seed our algorithms with an adequate initial program; can we search for better programs given an initial one rather than start from scratch? There can be multiple domain experts, each writing their own programs; how do we leverage multiple program sources that can differ in quality and even disagree with each other? These are all questions and challenges that can arise when programmatic deep learning is used in practice.

Expanding to more applications. We primarily test and evaluate our algorithms for behavior modeling, but in principle our methods can apply to other settings as well. For example, we use NEAR in Chapter 5 as a general-purpose tool to distill a neural encoder into a program. Would this strategy work for neural modules in other methods as well? In addition, what do programs and DSLs look like in other domains, such as images and text? Would the differentiability requirement of the DSL for NEAR be too much of a restriction? We expect that expanding to new applications will inevitably introduce new technical challenges.

Theoretical analysis. The strong empirical results presented in this thesis can be complemented with additional theoretical analysis. For example, can we model the noise of expert-written programs more explicitly to draw conclusions about the robustness of our algorithms? In Chapter 5, we extend the VAE framework to learn programmatically-interpretable representations, but at the cost of sub-optimal results from a pure likelihood/ELBO perspective. Can we formulate the learning objective from the ground up that formally encapsulates practically-oriented extensions of VAEs? We believe that theoretical analysis would not only strengthen our understanding of programmatic deep learning, but also allow domain experts to confidently rely on it in practice.

BIBLIOGRAPHY

Abbeel, Pieter and Andrew Y Ng (2004). "Apprenticeship learning via inverse reinforcement learning". In: *Proceedings of the twenty-first international conference on Machine learning*, p. 1.

Berman, Gordon J et al. (2014). "Mapping the stereotyped behaviour of freely moving fruit flies". In: *Journal of The Royal Society Interface* 11.99, p. 20140672.

Burgos-Artizzu, Xavier P et al. (2012). "Social behavior recognition in continuous video". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, pp. 1322–1329.

Chen, Ting et al. (2020). "A simple framework for contrastive learning of visual representations". In: *International conference on machine learning*. PMLR, pp. 1597–1607.

Chen, Xi, Yan Duan, et al. (2016). "Infogan: Interpretable representation learning by information maximizing generative adversarial nets". In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pp. 2180–2188.

Chen, Xi, Diederik P Kingma, et al. (2017). "Variational lossy autoencoder". In: *International Conference on Learning Representations*.

Cho, Kyunghyun et al. (2014). "On the properties of neural machine translation: Encoder-decoder approaches". In: *arXiv preprint arXiv:1409.1259*.

Chung, Junyoung et al. (2015). "A recurrent latent variable model for sequential data". In: *Advances in neural information processing systems*.

Creswell, Antonia et al. (2017). "Adversarial information factorization". In: *arXiv preprint arXiv:1711.05175*.

Dilokthanakul, Nat et al. (2016). "Deep unsupervised clustering with gaussian mixture variational autoencoders". In: *arXiv preprint arXiv:1611.02648*.

Dupont, Emilien (2018). "Learning disentangled joint continuous and discrete representations". In: *Proceedings of the 32nd Conference on Neural Information Processing Systems*.

Ellis, Kevin et al. (2020). "Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning". In: *arXiv preprint arXiv:2006.08381*.

Ellis, Kevin M et al. (2018). "Library learning for neurally-guided bayesian program induction". In: *Proceedings of the 32nd Conference on Neural Information Processing Systems*.

Eyjolfsdottir, Eyrun, Kristin Branson, et al. (2017). "Learning recurrent representations for hierarchical behavior modeling". In: *International Conference on Learning Representations*.

Eyjolfsdottir, Eyrun, Steve Branson, et al. (2014). "Detecting social actions of fruit flies". In: *European Conference on Computer Vision*. Springer, pp. 772–787.

Hausman, Karol et al. (2017). "Multi-modal imitation learning from unstructured demonstrations using generative adversarial nets". In: *arXiv preprint arXiv:1705.10479*.

Ho, Jonathan and Stefano Ermon (2016). "Generative adversarial imitation learning". In: *Advances in Neural Information Processing Systems*.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural Computation* 9.8, pp. 1735–1780.

Hong, Weizhe et al. (2015). "Automated measurement of mouse social behaviors using depth sensing, video tracking, and machine learning". In: *Proceedings of the National Academy of Sciences* 112.38, E5351–E5360.

Hrolenok, Brian, Byron Boots, and Tucker Hybinette Balch (2017). "Sampling beats fixed estimate predictors for cloning stochastic behavior in multiagent systems". In: *Thirty-First AAAI Conference on Artificial Intelligence*.

Johnson, Matthew J et al. (2016). "Composing graphical models with neural networks for structured representations and fast inference". In: *Advances in Neural Information Processing Systems*.

Kaiser, Lukasz et al. (2018). "Fast decoding in sequence models using discrete latent variables". In: *International Conference on Machine Learning*. PMLR, pp. 2390–2399.

Li, Yunzhu, Jiaming Song, and Stefano Ermon (2017). "Infogail: Interpretable imitation learning from visual demonstrations". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 3815–3825.

Luxem, Kevin et al. (2020). "Identifying behavioral structure from deep variational embeddings of animal motion". In: *BioRxiv*.

McQueen, Armand, Jenna Wiens, and John Guttag (2014). "Automatically recognizing on-ball screens". In: *2014 MIT Sloan Sports Analytics Conference*.

Oord, Aaron van den, Yazhe Li, and Oriol Vinyals (2018). "Representation learning with contrastive predictive coding". In: *arXiv preprint arXiv:1807.03748*.

Oord, Aaron van den, Oriol Vinyals, and Koray Kavukcuoglu (2017). "Neural discrete representation learning". In: *Proceedings of the 31st Conference on Neural Information Processing Systems*.

Pomerleau, Dean A. (1989). "ALVINN: An Autonomous Land Vehicle in a Neural Network". In: *Advances in Neural Information Processing Systems*.

Radford, Alec et al. (2018). "Improving language understanding by generative pre-training". In:

Ratner, Alexander J et al. (2016). "Data programming: Creating large training sets, quickly". In: vol. 29, pp. 3567–3575.

Ross, Stéphane, Geoffrey Gordon, and Drew Bagnell (2011). "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*, pp. 627–635.

Schroff, Florian, Dmitry Kalenichenko, and James Philbin (2015). "Facenet: A unified embedding for face recognition and clustering". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 815–823.

Segalin, Cristina et al. (2020). "The Mouse Action Recognition System (MARS): a software pipeline for automated analysis of social behaviors in mice". In: *BioRxiv*.

Sharma, Archit et al. (2020). "Dynamics-aware unsupervised discovery of skills". In: *International Conference on Learning Representations*.

Valkov, Lazar et al. (2018). "Houdini: Lifelong learning as program synthesis". In: *Proceedings of the 32nd Conference on Neural Information Processing Systems*.

Verma, Abhinav, Hoang M Le, et al. (2019). "Imitation-projected programmatic reinforcement learning". In: *Proceedings of the 33rd Conference on Neural Information Processing Systems*.

Verma, Abhinav, Vijayaraghavan Murali, et al. (2018). "Programmatically interpretable reinforcement learning". In: *International Conference on Machine Learning*. PMLR, pp. 5045–5054.

Vincent, Pascal et al. (2010). "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion." In: *Journal of machine learning research* 11.12.

Wang, Ziyu et al. (2017). "Robust imitation of diverse behaviors". In: *arXiv preprint arXiv:1707.02747*.

Zheng, Stephan, Yisong Yue, and Patrick Lucey (2016). "Generating long-term trajectories using deep hierarchical networks". In: *Advances in Neural Information Processing Systems*.

Ziebart, Brian D et al. (2008). "Maximum entropy inverse reinforcement learning." In: *Aaai*. Vol. 8, pp. 1433–1438.

Appendix A

APPENDIX TO CHAPTER 2

A.1 Boids Model Details

We generate 32,768 training and 8,192 test trajectories. Each agent's velocity is updated as:

$$\mathbf{v}_{t+1} = \beta \mathbf{v}_t + \beta(c_1 \mathbf{v}_{\text{coh}} + c_2 \mathbf{v}_{\text{sep}} + c_3 \mathbf{v}_{\text{ali}} + c_4 \mathbf{v}_{\text{ori}}), \tag{A.1}$$

where

- $\mathbf{v}_{\text{coh}}$ is the normalized cohesion vector towards an agent's local neighborhood (radius 0.9),
- $\mathbf{v}_{\text{sep}}$ is the normalized vector away from an agent's close neighborhood (radius 0.2),
- $\mathbf{v}_{\text{ali}}$ is the average velocity of other agents in a local neighborhood,
- $\mathbf{v}_{\text{ori}}$ is the normalized vector towards the origin,
- $(c_1, c_2, c_3, c_4) = (\pm 1, 0.1, 0.2, 1)$,
- β is sampled uniformly at random every 10 frames in range $[0.8, 1.4]$.

A.2 Maximizing Mutual Information

We ran experiments to see if we can learn meaningful macro-intents in a fully unsupervised fashion by maximizing the mutual information between macro-intent variables and trajectories τ. We use a VRAE-style model from (Fabius and Amersfoort, 2014) in which we encode an entire trajectory into a latent macro-intent variable $\mathbf{z}$, with the idea that $\mathbf{z}$ should encode global properties of the sequence. The corresponding ELBO is:

$$\mathcal{L}_1 = \mathbb{E}_{q_\phi(\mathbf{z}|\tau)} \left[\sum_{t=1}^{T} \sum_{k=1}^{K} \log \pi_{\theta_k}(\mathbf{s}_t^k \mid \mathbf{s}_{<t}, \mathbf{z}) \right] - D_{KL}\Big(q_\phi(\mathbf{z} \mid \tau) || p(\mathbf{z})\Big), \tag{A.2}$$

where $p(\mathbf{z})$ is the prior, $q_\phi(\mathbf{z} \mid \tau)$ is the encoder, and $\pi_{\theta_k}(\mathbf{s}_t^k \mid \mathbf{s}_{<t}, \mathbf{z})$ are policies per agent.

It is intractable to compute the mutual information between $\mathbf{z}$ and τ exactly, so we introduce a discriminator $q_\psi(\mathbf{z} \mid \tau)$ and use the following variational lower-bound of mutual information:

$$\mathcal{L}_2 = \mathcal{H}(\mathbf{z}) + \mathbb{E}_{\pi_\theta(\tau|\mathbf{z})}\Big[\mathbb{E}_{q_\phi(\mathbf{z}|\tau)}\big[\log q_\psi(\mathbf{z} \mid \tau)\big]\Big] \leq I(\tau; \mathbf{z}). \tag{A.3}$$

We jointly maximize $\mathcal{L}_1 + c\mathcal{L}_2$ wrt. model parameters (θ, ϕ, ψ), with $c = 1$ in our experiments.

Categorical vs. real-valued macro-intent z. When we train an 8-dimensional categorical macro-intent variable with a uniform prior (using Gumbel-Softmax trick (Jang, Gu, and Poole, 2017)), the average distribution from the encoder matches the discriminator but not the prior (Figure A.1). When we train a 2-dimensional real-valued macro-intent variable with a standard Gaussian prior, the learned model generates trajectories with limited variability as we vary the macro-intent variable (Figure A.2).

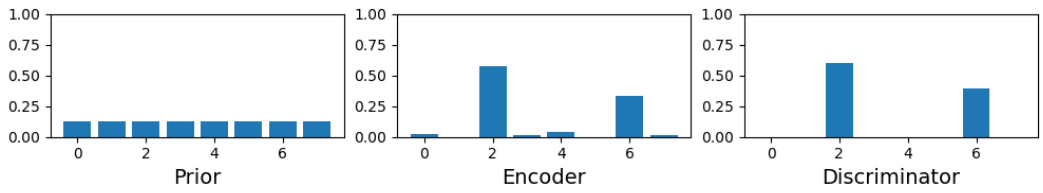


Figure A.1: Average distribution of 8-dimensional categorical macro-intent variable. The encoder and discriminator distributions match, but completely ignore the uniform prior distribution.

A.3 Programs for Macro-intents in Basketball

We define macro-intents in basketball by segmenting the left half-court into a 10 $\times$ 9 grid of 5ft $\times$ 5ft boxes (Figure 2.2). Algorithm 6 describes `Window25`, which computes macro-intents based on last positions in 25-timestep windows (`Window50` is similar). Algorithm 7 describes `Stationary`, which computes macro-intents based on stationary positions. For both, `Label-macro-intent`$(\mathbf{s}_t^k)$ returns the one-hot encoding of the box that contains the position $\mathbf{s}_t^k$.

References

Fabius, Otto and Joost R van Amersfoort (2014). "Variational recurrent auto-encoders". In: *ICLR workshop*.

Jang, Eric, Shixiang Gu, and Ben Poole (2017). "Categorical Reparameterization with Gumbel-Softmax". In: *ICLR*.

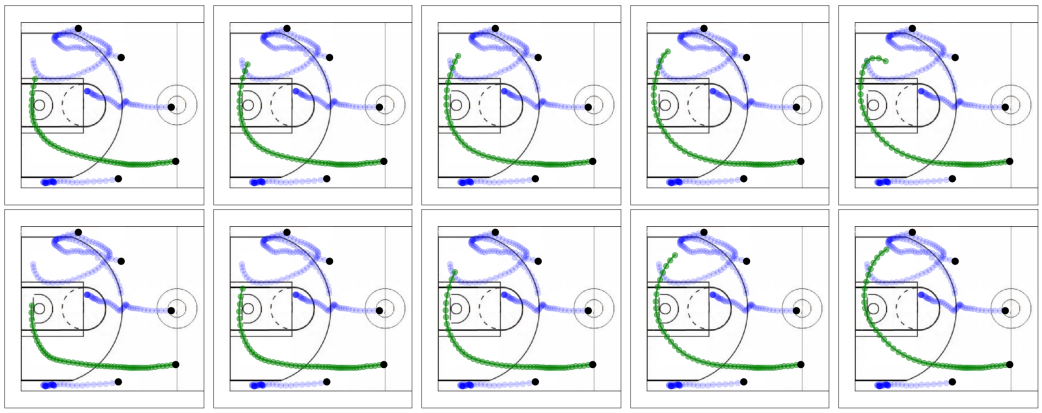


Figure A.2: Generated trajectories of green player conditioned on fixed blue players given various 2-dimensional macro-intent variables with a standard Gaussian prior. **Left to Right columns**: values of 1st dimension in $\{-1, -0.5, 0, 0.5, 1\}$. **Top row**: 2nd dimension equal to -0.5. **Bottom row**: 2nd dimension equal to 0.5. We see limited variability as we change the macro-intent variable.

Algorithm 6 Program that computes macro-intents in 25-timestep windows

procedure WINDOW25(τ) ▷ Trajectory with K players
  macro-intents $\mathbf{g} \leftarrow$ initialize array of size $(K, T, 90)$
  for $k = 1 \ldots K$ **do**
    $\mathbf{g}[k, T] \leftarrow$ LABEL-MACRO-INTENT($\mathbf{s}_T^k$) ▷ Last timestep
    for $t = T - 1 \ldots 1$ **do**
      if (t+1) mod 25 == 0 **then** ▷ End of window
        $\mathbf{g}[k, t] \leftarrow$ LABEL-MACRO-INTENT($\mathbf{s}_t^k$)
      else
        $\mathbf{g}[k, t] \leftarrow \mathbf{g}[k, t + 1]$
  return g

Algorithm 7 Program that computes macro-intents based on stationary positions

procedure STATIONARY(τ) ▷ Trajectory of K players
  macro-intents $\mathbf{g} \leftarrow$ initialize array of size $(K, T, 90)$
  for $k = 1 \ldots K$ **do**
    speed $\leftarrow$ compute speeds of player k in τ^k
    stationary $\leftarrow$ speed < threshold
    $\mathbf{g}[k, T] \leftarrow$ LABEL-MACRO-INTENT($\mathbf{s}_T^k$) ▷ Last timestep
    for $t = T - 1 \ldots 1$ **do**
      if stationary[t] and not stationary[t+1] **then** ▷ Player k moving
        $\mathbf{g}[k, t] \leftarrow$ LABEL-MACRO-INTENT($\mathbf{s}_t^k$)
      else ▷ Player k stationary
        $\mathbf{g}[k, t] \leftarrow \mathbf{g}[k, t + 1]$
  return g

Appendix B

APPENDIX TO CHAPTER 3

B.1 Baseline Policy Models

1) Conditional-TVAE (CTVAE). The conditional version of TVAEs optimizes:

$$\mathcal{L}^{\text{ctvae}}(\tau, \pi_\theta; q_\phi) = \mathbb{E}_{q_\phi(\mathbf{z}|\tau.\mathbf{y})}\left[\sum_{t=1}^{T} -\log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t, \mathbf{z}, \mathbf{y})\right] + D_{KL}\big(q_\phi(\mathbf{z}|\tau, \mathbf{y})||p(\mathbf{z})\big). \tag{B.1}$$

2) CTVAE with information factorization (CTVAE-info). (Creswell, Bharath, and Sengupta, 2017; Klys, Snell, and Zemel, 2018) augment conditional-VAE models with an auxiliary network $A_\psi(\mathbf{z})$ which is trained to predict the label $\mathbf{y}$ from $\mathbf{z}$, while the encoder q_ϕ is also trained to minimize the accuracy of A_ψ. This model *implicitly* maximizes self-consistency by removing the information correlated with $\mathbf{y}$ from $\mathbf{z}$, so that any information pertaining to $\mathbf{y}$ that the decoder needs for reconstruction must all come from $\mathbf{y}$. While this model was previously used for image generation, we extend it into the sequential domain:

$$\max_{\theta,\phi}\left(\mathbb{E}_{q_\phi(\mathbf{z}|\tau)}\left[\min_{\psi}\mathcal{L}^{\text{aux}}\big(A_\psi(\mathbf{z}), \mathbf{y}\big) + \sum_{t=1}^{T}\log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t, \mathbf{z}, \mathbf{y})\right] - D_{KL}\big(q_\phi(\mathbf{z}|\tau)||p(\mathbf{z})\big)\right). \tag{B.2}$$

Note that the encoder in (B.1) and (B.2) differ in that $q_\phi(\mathbf{z}|\tau)$ is no longer conditioned on the label $\mathbf{y}$.

3) CTVAE with mutual information maximization (CTVAE-mi). In addition to (B.1), we can also maximize the mutual information between labels and trajectories $I(\mathbf{y}; \tau)$. This quantity is hard to maximize directly, so instead we maximize the variational lower bound:

$$I(\mathbf{y}; \tau) \geq \mathbb{E}_{\mathbf{y}\sim p(\mathbf{y}), \tau\sim\pi_\theta(\cdot|\mathbf{z},\mathbf{y})}\left[\log r_\psi(\mathbf{y}|\tau)\right] + \mathcal{H}(\mathbf{y}), \tag{B.3}$$

where r_ψ approximates the true posterior $p(\mathbf{y}|\tau)$. In our setting, the prior over labels is known, so $\mathcal{H}(\mathbf{y})$ is a constant. Thus, the learning objective is:

$$\mathcal{L}^{\text{ctvae-mi}}(\tau, \pi_\theta; q_\phi) = \mathcal{L}^{\text{ctvae}}(\tau, \pi_\theta) + \mathbb{E}_{\mathbf{y}\sim p(\mathbf{y}), \tau\sim\pi_\theta(\cdot|\mathbf{z},\mathbf{y})}\left[-\log r_\psi(\mathbf{y}|\tau)\right]. \tag{B.4}$$

Optimizing (B.4) also requires collecting rollouts with the current policy, so similarly we also pretrain and fine-tune a dynamics model P_φ. This baseline can be interpreted as a supervised analogue of unsupervised models that maximize mutual information in (Li, Song, and Ermon, 2017; Hausman et al., 2017).

B.2 Stochastic Dynamics Function

If the dynamics function P of the environment is stochastic, we modify our approach in Algorithm 2 by changing the form of our dynamics model. We can model the change in state as a Gaussian distribution and minimize the negative log-likelihood:

$$\varphi_\mu^*, \varphi_\sigma^* = \arg\min_{\varphi_\mu, \varphi_\mu} \mathbb{E}_{\tau\sim\mathcal{D}} \sum_{t=1}^{T} -\log p(\Delta_t; \mu_t, \sigma_t), \tag{B.5}$$

where $\Delta_t = \mathbf{s}_{t+1} - \mathbf{s}_t$, $\mu_t = P_{\varphi_\mu}(\mathbf{s}_t, \mathbf{a}_t)$, $\sigma_t = P_{\varphi_\sigma}(\mathbf{s}_t, \mathbf{a}_t)$, and P_{φ_μ}, P_{φ_σ} are neural networks that can share weights. We can sample a change in state during rollouts using the reparametrization trick (Kingma and Welling, 2014), which allows us to backpropagate through the dynamics model during training.

B.3 Experiment Details and Hyperparameters

Dataset details. See Table B.1. Basketball trajectories are collected from tracking real players in the NBA. Figure B.2 shows the distribution of basketball programs applied on the training set. For Cheetah, we train 125 policies using PPO (Schulman et al., 2017) to run forwards at speeds ranging from 0 to 4 (m/s). We collect 25 trajectories per policy by sampling actions from the policy. We use (Kostrikov, 2018) to interface with (Tassa et al., 2018). Figure B.3 shows the distributions of Cheetah programs applied on the training set.

Hyperparameters. See Table B.2 for training hyperparameters and Table B.3 for model hyperparameters.

	$\lvert\mathcal{S}\rvert$	$\lvert\mathcal{A}\rvert$	T	N_{train}	N_{test}	frequency (Hz)
Basketball	2	2	24	520,015	67,320	3
Cheetah	18	6	200	2,500	625	40

Table B.1: Dataset parameters for basketball and Cheetah environments.

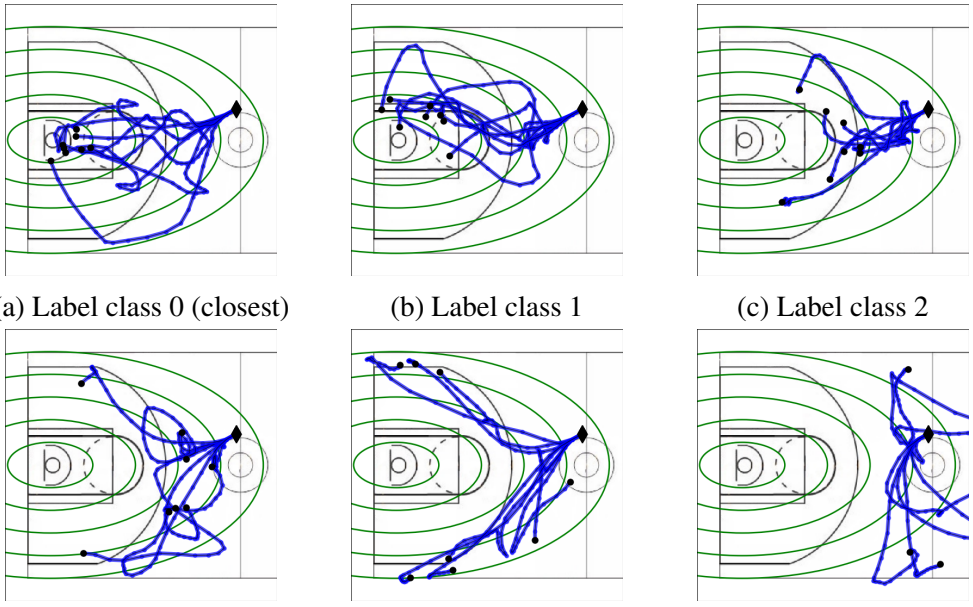
B.4 Additional Experiment Figures and Tables

	bs	b	n_{dynamics}	n_{label}	n_{policy}	n_{collect}	n_{env}	lr
Basketball	128	4,063	$10 \cdot b$	$20 \cdot b$	$30 \cdot b$	128	0	$2 \cdot 10^{-4}$
Cheetah	16	157	$50 \cdot b$	$20 \cdot b$	$60 \cdot b$	16	1	10^{-3}

Table B.2: Hyperparameters for Algorithm 2. bs is the batch size, b is the number of batches to see all trajectories in the dataset once, and lr is the learning rate. We also use L_2 regularization of 10^{-5} for training the dynamics model P_φ.

	$\mathbf{z}$-dim	q_ϕ GRU	C^λ_ψ GRU	π_θ GRU	π_θ sizes	P_φ sizes
Basketball	4	128	128	128	(128,128)	(128,128)
Cheetah	8	200	200	-	(200,200)	(500,500)

Table B.3: Model parameters for Basketball and Cheetah environments.



(a) Label class 0 (closest) (b) Label class 1 (c) Label class 2

(d) Label class 3 (e) Label class 4 (f) Label class 5 (farthest)

Figure B.1: Rollouts from our policy calibrated to `Destination(basket)` with 6 classes. The 5 green boundaries divide the court into 6 regions, each corresponding to a label class. The label indicates the target region of a trajectory's final position (•). This policy achieves a style-consistency of 0.93, as indicated in Table B.4c. Note that the initial position (⧫) is the same as in Figures 3.3 and 3.4 for comparison, but in general we sample an initial position from the prior $p(\mathbf{y})$ to compute style-consistency.

Model	Speed			Disp.			Dest.			Dir.			Curve		
CTVAE	82	83	85	71	72	74	81	82	82	76	77	80	60	61	62
CTVAE-info	**84**	84	87	69	71	74	78	79	83	71	72	74	60	60	62
CTVAE-mi	84	86	87	71	74	74	80	82	84	75	77	78	58	72	74
CTVAE-style	34	**95**	**97**	**89**	**96**	**97**	**91**	**97**	**98**	**96**	**97**	**98**	**77**	**81**	**83**

(a) Style-consistency wrt. single styles of 3 classes (roughly uniform distributions).

Model	2 classes			3 classes			4 classes			6 classes			8 classes		
CTVAE	91	92	93	79	83	84	**76**	79	79	**68**	70	72	64	66	69
CTVAE-info	90	90	92	**83**	83	85	75	76	77	68	70	72	60	63	67
CTVAE-mi	90	92	93	81	84	86	75	77	80	66	70	72	62	62	67
CTVAE-style	**98**	**99**	**99**	15	**98**	**99**	15	**96**	**96**	02	**92**	**94**	**80**	**90**	**93**

(b) Style-consistency wrt. `Displacement` of up to 8 classes (roughly uniform distributions).

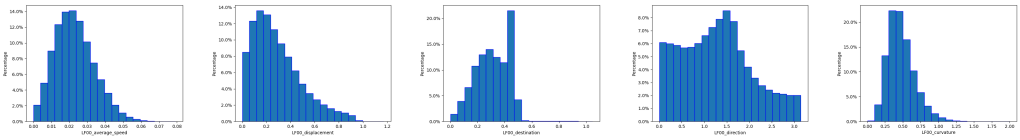
Model	2 classes			3 classes			4 classes			6 classes		
CTVAE	86	87	87	80	82	83	**76**	78	79	70	74	77
CTVAE-info	83	87	88	79	81	83	73	75	78	71	77	78
CTVAE-mi	86	88	88	**80**	81	84	71	74	79	**73**	76	78
CTVAE-style	**97**	**98**	**99**	68	**97**	**98**	35	**89**	**95**	67	**84**	**93**

(c) Style-consistency wrt. `Destination(basket)` with up to 6 classes (non-uniform distributions).

Model	2 styles, 3 classes			3 styles, 3 classes			4 styles, 3 classes			5 styles, 3 classes			5 styles, 4 classes		
CTVAE	67	71	73	58	58	62	49	50	52	27	37	35	**20**	21	22
CTVAE-info	68	69	70	54	58	59	48	51	54	28	32	35	18	21	23
CTVAE-mi	71	72	73	48	56	61	45	51	52	16	30	31	18	21	23
CTVAE-style	**92**	**93**	**94**	**86**	**88**	**90**	**62**	**88**	**88**	**66**	**75**	**80**	11	**55**	**77**

(d) Style-consistency wrt. multiple styles simultaneously.

Table B.4: [min, median, max] style-consistency ($\times 10^{-2}$, 5 seeds) of policies evaluated with 4,000 basketball rollouts each. CTVAE-style policies significantly outperform baselines in all experiments and are calibrated at almost maximal style-consistency for 4/5 programs. We note some rare failure cases with our approach, which we leave as a direction for improvement for future work.



(a) Speed (b) Displacement (c) Destination (d) Direction (e) Curvature

Figure B.2: Histogram of basketball programs applied on the training set (before applying thresholds). Basketball trajectories are collected from tracking real players in the NBA.

Model	Speed			Torso-height			B-foot-height			F-foot-height		
CTVAE	53	59	62	62	63	70	61	68	73	63	68	72
CTVAE-info	56	57	61	62	63	72	58	65	72	63	66	69
CTVAE-mi	53	60	62	62	65	70	60	65	70	66	70	73
CTVAE-style	**68**	**79**	**81**	**79**	**80**	**84**	**77**	**80**	**88**	**74**	**77**	**80**

(a) Style-consistency wrt. single styles of 2 classes (roughly uniform distributions).

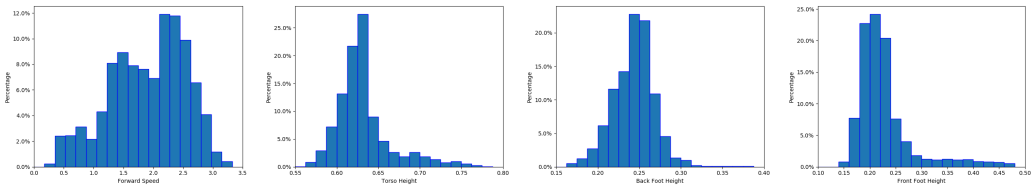
Model	3 classes			4 classes		
CTVAE	41	45	49	35	37	41
CTVAE-info	47	49	52	36	39	42
CTVAE-mi	47	48	53	36	37	38
CTVAE-style	**59**	**59**	**65**	**42**	**51**	**60**

(b) Style-consistency wrt. `Speed` with varying # of classes (non-uniform distributions).

Model	2 styles, 2 classes			3 styles, 2 classes		
CTVAE	39	41	43	25	28	29
CTVAE-info	39	41	46	25	27	30
CTVAE-mi	34	40	48	27	28	31
CTVAE-style	**43**	**54**	**60**	**38**	**40**	**52**

(c) Style-consistency wrt. multiple styles simultaneously.

Table B.5: [min, median, max] style-consistency ($\times 10^{-2}$, 5 seeds) of policies evaluated with 500 Cheetah rollouts each. CTVAE-style policies consistently outperform all baselines, but we note that there is still room for improvement (to reach 100% style-consistency).



(a) Speed (b) Torso-height (c) Back-foot-height (d) Front-foot-height

Figure B.3: Histogram of Cheetah programs applied on the training set (before applying thresholds). Note that `Speed` is the most diverse behavior because we pre-trained the policies to achieve various speeds when collecting demonstrations, similar to (Wang et al., 2017). For more diversity with respect to other behaviors, we can also incorporate a target behavior as part of the reward when pre-training Cheetah policies.

Model	Speed	Disp.	Dest.	Dir.	Curve
CTVAE	83.4 ± 1.2	72.4 ± **1.4**	81.9 ± **0.6**	77.7 ± 1.3	61.0 ± 1.0
CTVAE-info	85.0 ± **1.2**	71.2 ± 1.9	80.1 ± 1.8	72.3 ± 1.1	60.2 ± **0.8**
CTVAE-mi	**85.8** ± 1.3	72.8 ± 1.5	82.2 ± 1.4	76.9 ± 1.1	68.6 ± 6.4
CTVAE-style	72.1 ± 33.	**94.6** ± 3.1	**95.0** ± 3.7	**96.8** ± **0.7**	**79.6** ± 2.7

(a) Style-consistency wrt. single styles of 3 classes (roughly uniform distributions).

Model	2 classes	3 classes	4 classes	6 classes	8 classes
CTVAE	92.1 ± 0.9	82.4 ± 2.4	78.0 ± 1.4	69.9 ± **1.4**	66.0 ± **2.0**
CTVAE-info	90.5 ± 0.9	**83.6** ± **1.0**	75.9 ± **0.9**	**70.2** ± 1.6	63.4 ± 2.9
CTVAE-mi	91.6 ± 1.2	83.5 ± 2.1	77.6 ± 2.5	68.8 ± 2.5	63.7 ± 2.3
CTVAE-style	**98.7** ± **0.4**	81.4 ± 36.9	**79.3** ± 35.9	68.1 ± 40.0	**88.2** ± 5.1

(b) Style-consistency wrt. `Displacement` of up to 8 classes (non-uniform distributions).

Model	2 classes	3 classes	4 classes	6 classes
CTVAE	86.6 ± **0.6**	81.6 ± **1.3**	**77.4** ± **1.5**	74.0 ± 2.6
CTVAE-info	86.2 ± 1.7	81.1 ± 1.4	75.3 ± 2.5	75.3 ± 3.3
CTVAE-mi	87.3 ± 0.9	81.6 ± 1.6	74.3 ± 3.1	75.8 ± **2.1**
CTVAE-style	**98.1** ± 0.8	**88.2** ± 13.6	77.0 ± 24.1	**82.6** ± 11.3

(c) Style-consistency wrt. `Destination(basket)` of up to 6 classes (non-uniform distributions).

Model	2 styles, 3 classes	3 styles, 3 classes	4 styles, 3 classes	5 styles, 3 classes	5 styles, 4 classes
CTVAE	70.5 ± 2.1	58.9 ± **1.5**	50.4 ± **1.4**	31.6 ± 2.8	**20.8** ± 1.0
CTVAE-info	69.0 ± 0.9	57.5 ± 2.0	50.5 ± 2.3	31.4 ± **2.5**	20.6 ± 2.0
CTVAE-mi	71.8 ± **0.7**	53.8 ± 5.9	50.2 ± 2.7	26.9 ± 6.3	20.7 ± 1.9
CTVAE-style	**92.8** ± 1.0	**88.3** ± 1.7	**81.7** ± 11.0	**73.9** ± 5.4	**50.3** ± 24.7

(d) Style-consistency wrt. multiple styles simultaneously.

Table B.6: Mean and standard deviation style-consistency ($\times 10^{-2}$, 5 seeds) of policies evaluated with 4,000 basketball rollouts each. CTVAE-style policies generally outperform baselines. Lower mean style-consistency (and large standard deviation) for CTVAE-style is often due to failure cases, as can be seen from the minimum style-consistency values we report in Table B.4. Understanding the causes of these failure cases and improving the algorithm's stability are possible directions for future work.

Model	Speed	Torso-height	B-foot-height	F-foot-height
CTVAE	57.4 ± 3.9	64.4 ± 3.1	67.4 ± 4.2	68.5 ± 3.7
CTVAE-info	58.3 ± **2.1**	65.0 ± 4.2	64.1 ± 5.4	66.1 ± 2.7
CTVAE-mi	58.4 ± 3.9	65.7 ± 3.2	65.0 ± **3.6**	69.9 ± 2.6
CTVAE-style	**77.0** ± 5.3	**81.0** ± **2.2**	**81.9** ± 5.4	**77.2** ± **2.4**

(a) Style-consistency wrt. single styles of 2 classes (roughly uniform distributions).

Model	3 classes	4 classes
CTVAE	45.2 ± 3.2	37.8 ± 2.9
CTVAE-info	49.2 ± **1.8**	39.3 ± 2.8
CTVAE-mi	49.1 ± 2.2	36.8 ± **1.0**
CTVAE-style	**60.8** ± 2.9	**51.3** ± 7.8

(b) Style-consistency wrt. `Speed` with varying # of classes (non-uniform distributions).

Model	2 styles, 2 classes	3 styles, 2 classes
CTVAE	40.9 ± **1.6**	27.2 ± 1.9
CTVAE-info	41.8 ± 2.3	27.8 ± 2.2
CTVAE-mi	40.7 ± 4.9	28.5 ± **1.6**
CTVAE-style	**52.6** ± 6.1	**42.8** ± 5.8

(c) Style-consistency wrt. multiple styles simultaneously.

Table B.7: Mean and standard deviation style-consistency ($\times 10^{-2}$, 5 seeds) of policies evaluated with 500 Cheetah rollouts each. CTVAE-style policies consistently outperform all baselines, but we note that there is still room for improvement (to reach 100% style-consistency).

	Speed		Torso-height		B-foot-height		F-foot-height	
Model	NLD	SC	NLD	SC	NLD	SC	NLD	SC
CTVAE-style	-0.28	**79**	-0.24	80	-0.16	**80**	-0.22	**77**
CTVAE-style+	-0.49	70	-0.42	**83**	-0.36	**80**	-0.42	74

Table B.8: We report the median negative log-density per timestep (lower is better) and style-consistency (higher is better) of CTVAE-style policies for Cheetah (5 seeds). The first row corresponds to experiments in Tables 3.1 and B.5a, and the second row corresponds to the same experiments with 50% more training iterations. The KL-divergence in the two sets of experiments are roughly the same. Although imitation quality improves, style-consistency can sometimes degrade (e.g., `Speed`, `Front-foot-height`), indicating a possible trade-off between imitation quality and style-consistency.

	Style-consistency ↑					
Model	**Min**	-	**Median**	-	**Max**	NLD ↓
RNN	79	79	80	81	81	-7.7
RNN-style	81	86	91	95	98	-7.6
CTVAE	81	82	82	82	82	**-8.0**
CTVAE-style	**91**	**92**	**97**	**98**	**98**	-7.8

Table B.9: Comparing style-consistency ($\times 10^{-2}$) between RNN and CTVAE policy models for `Destination(basket)` in basketball. The style-consistency for 5 seeds are listed in increasing order. Our algorithm improves style-consistency for both policy models at the cost of a slight degradation in imitation quality. In general, CTVAE performs better than RNN in both style-consistency and imitation quality.

	Speed	**Disp.**	**Dest.**	**Dir.**	**Cure**
$\mathcal{L}^{\text{label}}$	3.96 ± 0.33	4.58 ± 0.20	1.61 ± 0.18	3.19 ± 0.25	28.31 ± 0.95

(a) Basketball programs for experiments in Experiment 1.

	Speed	**Torso-height**	**B-foot-height**	**F-foot-height**
$\mathcal{L}^{\text{label}}$	3.24 ± 0.83	15.87 ± 1.78	17.25 ± 0.73	14.75 ± 0.74

(b) Cheetah programs for experiments in Experiment 1.

Table B.10: Mean and standard deviation cross-entropy loss ($\mathcal{L}^{\text{label}}, \times 10^{-2}$) over 5 seeds of learned label approximators $C^{\lambda}_{\psi^*}$ on test trajectories after n^{label} training iterations for experiments in Experiment 1. $C^{\lambda}_{\psi^*}$ is only used during training; when computing style-consistency for our quantitative results, we use original programs λ.

	P_φ test error
Basketball	$1.47 \pm 0.59 (\times 10^{-7})$
Cheetah	$1.93 \pm 0.08 (\times 10^{-2})$

Table B.11: Average mean-squared error of dynamics model P_φ per timestep per dimension on test data after training for n^{dynamics} iterations.

	Basketball				
noise	**Speed**	**Disp.**	**Dest.**	**Dir.**	**Curve**
σ	5.20	6.18	7.46	5.36	5.88
2σ	10.33	12.24	15.54	10.93	11.66
3σ	15.36	18.08	23.46	16.78	17.24
4σ	20.10	23.47	30.10	22.56	22.52
σ value	0.001	0.02	0.02	0.1	0.02

Table B.12: **Label disagreement (%) of noisy programs:** For each of the Basketball programs with 3 classes in Table 3.1, we consider noisy versions where we inject Gaussian noise with mean 0 and standard deviation $c \cdot \sigma$ for $c \in \{1, 2, 3, 4\}$ before applying thresholds to obtain label classes. This table shows the label disagreement between noisy and true programs over trajectories in the training set. The last row shows the σ value used for each program.

	Basketball				
noise	**Speed**	**Disp.**	**Dest.**	**Dir.**	**Curve**
σ	2.78	3.21	3.70	3.71	3.16
2σ	5.59	7.88	9.75	8.63	4.46
3σ	9.71	15.37	16.38	12.39	6.34
4σ	11.63	20.54	21.11	19.98	12.41

Table B.13: **Relative *decrease* in style-consistency** when training with noisy programs: (%, median over 5 seeds) Using the noisy programs in Table B.12, we train CTVAE-style models and evaluate style-consistency using the true programs without noise. This table shows the percentage decrease in style-consistency relative to when there is no noise in Table 3.1. Comparing with the label disagreement in Table B.12, we see that the relative decrease in style-consistency generally scales linearly with the label disagreement between noisy and true programs.

References

Creswell, Antonia, Anil A. Bharath, and Biswa Sengupta (2017). "Adversarial Information Factorization". In: *arXiv preprint arXiv:1711.05175*.

Hausman, Karol et al. (2017). "Multi-Modal Imitation Learning from Unstructured Demonstrations using Generative Adversarial Nets". In: *Neural Information Processing Systems (NeurIPS)*.

Kingma, Diederik P and Max Welling (2014). "Auto-encoding variational bayes". In: *International Conference on Learning Representations (ICLR)*.

Klys, Jack, Jake Snell, and Richard S. Zemel (2018). "Learning Latent Subspaces in Variational Autoencoders". In: *Neural Information Processing Systems (NeurIPS)*.

Kostrikov, Ilya (2018). *PyTorch Implementations of Reinforcement Learning Algorithms*. `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail`.

Li, Yunzhu, Jiaming Song, and Stefano Ermon (2017). "InfoGAIL: Interpretable Imitation Learning from Visual Demonstrations". In: *Neural Information Processing Systems (NeurIPS)*.

Schulman, John et al. (2017). "Proximal Policy Optimization Algorithms". In: *arXiv preprint arXiv:1707.06347*.

Tassa, Yuval et al. (2018). "DeepMind Control Suite". In: *arXiv preprint arXiv:1801.00690*.

Wang, Ziyu et al. (2017). "Robust Imitation of Diverse Behaviors". In: *Neural Information Processing Systems (NeurIPS)*.

Appendix C

APPENDIX TO CHAPTER 4

C.1 Iterative Deepening Depth-First-Search

In Algorithm 8, we provide the pseudocode for the IDS-BB algorithm introduced in the main text. This algorithm is a heuristic-guided depth-first search with three key characteristics: (1) the search depth is iteratively increased; (2) the search is ordered using a function $f(u)$ as in A*, and (3) branch-and-bound is used to prune unprofitable parts of the search space. We find that the use of iterative deepening in the program learning setting is useful in that it prioritizes searching shallower and less parsimonious programs early on in the search process.

Algorithm 8 Iterative Deepening Depth-First-Search

Input: initial depth d_{initial}, max depth $d_{\max}$
Initialize *frontier*, *nextfrontier* as priority-queues for tuples $(u, f(u), d_u)$ for nodes $u \in \mathcal{G}$, prioritizing smaller f-scores first
$(current, best, f_{\min}, d_{\text{iter}}) := (None, None, \infty, d_{\text{initial}})$
frontier.push($(u_0, \infty, 0)$)
while *frontier* $\neq \emptyset$ **do**
    if *current* is $None$ **then**
        $(current, f_{\text{current}}, d_{\text{current}}) :=$ *frontier*.pop()
    if *current* is a goal node **then**
        if $f_{\text{current}} < f_{\min}$ **then**
            $f_{\min} := f_{\text{current}}$
            $best :=$ *current*
        current $:= None$
    else if $d_{\text{current}} > d_{\text{iter}}$ **then**
        current $:= None$
    else
        Compute f-scores for all children of *current*
        Set *current* equal to child with smallest f-score
        if $d_{\text{current}} \leq d_{\max}$ **then**
            for child w of *current* **do**
                frontier.push($(w, f(w), d_w)$)
        if *frontier* $= \emptyset$ **then**
            frontier := *nextfrontier*
            nextfrontier $:= \emptyset$
            $d_{\text{iter}} = d_{\text{iter}} + 1$
return $best, f_{\min}$

C.2 Hyperparameter Details

In tables C.1, C.2, C.3, C.4, and C.5 we present the hyperparameters used in our implementation for all baselines. Usage of each hyperparameter can be found in our codebase. We elaborate below on hyperparameters specific to our contribution, namely A*-NEAR and IDS-BB-NEAR.

In A*-NEAR and IDS-BB-NEAR, we allow for a number of hyperparameters to be used that can additionally speed up our search. To improve efficiency, we allow for the frontier in these searches to be bounded by a constant size. In doing so, we sacrifice the completeness guarantees discussed in the main text in exchange for additional efficiency. We also allow for a scalar performance multiplier, which is a number greater than zero, that is applied to each node in the frontier when a goal node is found. The nodes on the frontier must have a lower cost than the goal node after this performance multiplier is applied; otherwise, they are pruned from the frontier in the case of branch-and-bound. When considering non-goal nodes, this multiplier is not applied. We introduce an additional parameter that decreases this performance multiplier as nodes get farther from the source node, i.e., become more complete programs. We also decrease the number of units given to a neural network within a *neural program approximation* as nodes get further from the source node, with the intuition that neural program induction done in a more complete program will likely have less complex behavior to induce. We also allow for the branching factor of all nodes in the graph to be bounded to a user-specified width in order to bound the combinatorial explosion of program space. This constraint comes at the expected sacrifice of completeness in our program search, given that potentially optimal paths are arbitrarily not considered.

When searching over these hyperparameters, we noted that the the performance of the neural program approximations as a heuristic required a balance in their levels of accuracy with respect to the given optimization objective. If the approximations are heavily underparameterized, the NEAR heuristic may not reach the same level of expressivity as the DSL, and as a result, poor performance can lead to the heuristic being inadmissible. However, if the approximations are so expressive that they are able to achieve near-perfect accuracy on the task, admissibility will still hold, but the resulting heuristic will become uninformative and inefficient. For example, a NEAR heuristic that only returns 0 will remain admissible, but the performance of A*-NEAR in this case will be no better than breadth-first search. With this in mind, we searched for a set of hyperparameters that yielded an informative level of

performance.

In our experiments, we show that using the aforementioned approximative hyperparameters allows for an accelerated search while maintaining strong empirical results with our NEAR-guided search algorithms.

	max depth	init. # units	min # units	max # children	penalty	β
CRIM13-sniff	10	15	6	8	0.01	1.0
CRIM13-other	10	15	6	8	0.01	1.0
Fly-vs.-Fly	6	25	10	6	0.01	1.0
Bball-ballhandler	8	16	4	8	0.01	1.0

Table C.1: Hyperparameters for constructing graph $\mathcal{G}$.

	# LSTM units	# epochs	learning rate	batch size
CRIM13-sniff	100	50	0.001	50
CRIM13-other	100	50	0.001	50
Fly-vs.-Fly	80	40	0.00025	30
Bball-ballhandler	64	15	0.01	50

Table C.2: Training hyperparameters for RNN baseline.

	# neural epochs	# symbolic epochs	learning rate	batch size
CRIM13-sniff	6	15	0.001	50
CRIM13-other	6	15	0.001	50
Fly-vs.-Fly	6	25	0.00025	30
Bball-ballhandler	4	6	0.02	50

Table C.3: Training hyperparameters for all program learning algorithms. The # neural epochs hyperparameter refers only to the number of epochs that neural program approximations were trained in NEAR strategies.

	A*-NEAR	IDS-BB-NEAR			
	frontier size	frontier size	initial depth	depth bias	perf. multiplier
CRIM13-sniff	8	8	5	0.95	0.975
CRIM13-other	8	8	5	0.95	0.975
Fly-vs.Fly	10	10	4	0.9	0.95
Bball-ballhandler	400	30	3	1.0	1.0

Table C.4: Additional hyperparameters for A*-NEAR and IDS-BB-NEAR.

	MC(TS)	Enum.	Genetic					
	samples /step	max # prog.	pop. size	select. size	# gens	total # evals	mutate prob.	enum. depth
CRIM13-sniff	50	300	15	8	20	100	0.1	5
CRIM13-other	50	300	15	8	20	100	0.1	5
Fly-vs.Fly	25	100	20	10	10	10	0.1	6
Bball-ballhandler	150	1200	100	50	10	1000	0.01	7

Table C.5: Additional hyperparameters for other program learning baselines

C.3 Additional Details on Experimental Domains

Fly-v.-Fly

The *Fly-vs.-Fly* dataset (Eyjolfsdottir et al., 2014) tracks a pair of flies and their actions as they interact in different contexts. Each timestep is represented by a 53-dimensional feature vector including 17 features outlining the fly's position and orientation along with 36 position-invariant features, such as linear and angular velocities. Our task in this domain is that of *bout-level classification*, where we are tasked to classify a given trajectory of timesteps to a corresponding single action taking place. Of the three datasets within *Fly-vs.-Fly*, we use the *Aggression* and *Boy-meets-Boy* datasets and classify trajectories over the 7 labeled actions displaying aggressive, threatening, and nonthreatening behaviors in these two datasets. We omit the use of the *Courtship* dataset for our classification task, primarily due to the heavily skewed trajectories in this dataset that vary highly in length and action type from the *Aggression* and *Boy-meets-Boy* datasets. Full details on these datasets, as well as where to download them, can be found in (Eyjolfsdottir et al., 2014). To ensure a desired balance in our training set, we limit the length of trajectories to 300 timesteps, and break up trajectories that exceed this length into separate trajectories with the same action label for data augmentation. Our training dataset has 5339 trajectories, our validation set has 594 trajectories, and our test set has 1048 trajectories. The average length of a trajectory is 42.06 timesteps.

Training details of Fly-v.-Fly baselines. For all of our program synthesis baselines , we used the Adam (Kingma and Ba, 2014) optimizer and cross-entropy loss. Each synthesis baseline was run on an Intel 4.9-GHz i7 CPU with 8 cores, equipped with an NVIDIA RTX 2070 GPU w/ 2304 CUDA cores.

CRIM13

The CRIM13 dataset studies the social behavior of a pair of mice annotated each frame by behavior experts (Burgos-Artizzu et al., 2012) at 25Hz. The interaction between a resident mouse and an intruder mouse, which is introduced to the cage of the resident, is recorded. Each mice is tracked by one keypoint and a 19 dimensional feature vector based on this tracking data is provided at each frame. The feature vector consists of features such as velocity, acceleration, distance between mice, angle and angle changes. Our task in this domain is *sequence classification*: we classify each frame with a behavior label from CRIM13. Every frame is labelled with one of 12 actions, or "other". The "other" class corresponds to cases where no action of interest is occurring. Here, we focus on two binary classification tasks: other vs. rest, and sniff vs. rest. The first task, other vs. rest, corresponds to labeling whether there is an action of interest in the frame. The second task, sniff vs. rest, corresponds to whether the resident mouse is sniffing any part of the intruder mouse. These two tasks are chosen such that the RNN baseline has reasonable performance only using the tracked keypoint features of the mice. We split the train set in (Burgos-Artizzu et al., 2012) at the video level into our train and validation set, and we present test set results on the same set as (Burgos-Artizzu et al., 2012). Each video is split into sequences of 100 frames. There are 12404 training trajectories, 3077 validation trajectories, and 2953 test trajectories.

We observed higher variance in F1 score for the CRIM13-sniff class in Table C.7, as compared to the other experiments. For this particular class, due to the high variance of both baseline and NEAR runs, we would like to note the importance of repeating runs.

Training details of CRIM13 baselines. All CRIM13 baselines training uses the Adam (Kingma and Ba, 2014) optimizer and cross-entropy loss. In the loss for sniff vs. rest, the sniff class is weighted by 1.5. Each synthesis baseline was run on an Intel 2.2-GHz Xeon CPU with 4 cores, equipped with an NVIDIA Tesla P100 GPU with 3584 CUDA cores.

Basketball

The basketball data tracks player positions (xy-coordinates on court) from real professional games. We used the processed version from (Yue et al., 2014), which includes trajectories over 8 seconds (3 Hz in our case of sequence length 25) centered on the left half-court. Among the offensive and defensive teams, players are ordered

based on their relative positions. Labels for the ballhandler were extracted with a labeling function written by a domain expert. See Table C.6 for full details of this dataset.

Training details of Basketball baselines. All Basketball experiments use Adam (Kingma and Ba, 2014) and optimize cross-entropy loss. Each synthesis baseline was run on an Intel 3.6-GHz i7-7700 CPU with 4 cores, equipped with an NVIDIA GTX 1080 Ti GPU with 3584 CUDA cores.

	state dim	label dim	max seq. len.	# train	# valid	# test
CRIM13-sniff	19	2	100	12404	3007	2953
CRIM13-other	19	2	100	12404	3007	2953
Fly-vs.-Fly	53	7	300	5339	594	1048
Bball-ballhandler	22	6	25	18000	2801	2893

Table C.6: Dataset details.

	CRIM13-sniff			**CRIM13-other**			**Fly-vs.-Fly**			**Basketball**		
	Acc.	F1	*d*	Acc.	F1	*d*	Acc.	F1	*d*	Acc.	F1	*d*
Enum.	.024	.105	1	.036	.011	1	.013	.012	0	.009	.009	0.6
MC	.013	.127	1.7	.088	.031	0.6	.028	.018	2	.012	.012	0.6
MCTS	.047	.076	0	.103	.036	0	.008	.009	0.94	.003	.002	0
Genetic	.003	.015	0.6	.005	.004	1.7	.028	.030	1	.016	.019	0.6
IDDFS-NEAR	.021	.056	2	.006	.005	0.6	.023	.016	0	.006	.006	0
A*-NEAR	.026	.114	1.7	.030	.010	2.1	.003	.004	0	.034	.034	0
RNN	.008	.019	-	.005	.002	-	.006	.005	-	.001	.001	-

Table C.7: Standard Deviations of accuracy, F1-score, and program depth *d* of learned programs (3 trials).

References

Burgos-Artizzu, Xavier P et al. (2012). "Social behavior recognition in continuous video". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, pp. 1322–1329.

Eyjolfsdottir, Eyrun et al. (2014). "Detecting social actions of fruit flies". In: *European Conference on Computer Vision*. Springer, pp. 772–787.

Kingma, Diederik and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980*.

Yue, Yisong et al. (2014). "Learning fine-grained spatial models for dynamic sports play prediction". In: *ICDM*.

Appendix D

APPENDIX TO CHAPTER 5

D.1 Additional Results

Table D.1 contains the standard deviations of the results in Table 5.1 of the main paper.

Table D.2 contains the median ELBO of our baselines and our neurosymbolic encoders. We find that our symbolic encoders are comparable with our baselines. This is expected: since we are imposing additional constraints on the encoder (a program with a bounded depth), we would not expect the variational approximation to be better than an encoder without these constraints (fully-neural encoder). In general, obtaining better or more semantically-meaningful cluster assignments can come at the cost of a smaller ELBO. For example, we find that introducing a clustering loss to the TVAE can result in better metrics, but lower ELBO as well.

Model	**CalMS21**			**Basketball**		
	Purity	NMI	RI	Purity	NMI	RI
TVAE	.002	.011	.001	.049	.012	.008
TVAE+KMeans loss	.001	.002	.001	.006	.001	.001
JointVAE	.000	.003	.022	.037	.020	.004
VQ-TVAE	.005	.004	.016	.042	.022	.014
Beta-TVAE	.001	.001	.001	.124	.140	.088
Ours (1 program)	.026	.056	.035	.039	.014	.001
Ours (2 programs)	.017	.051	.019	.053	.020	.018
Ours (3 programs)	.088	.075	.030	.007	.002	.002

Table D.1: Standard deviation of purity, NMI, and RI on CalMS21 and Basketball compared to human-annotated labels (3 runs). Random assignment metrics have standard deviation close to 0.

D.2 Implementation Details

Hyperparameters

The hyperparameters for our approach are in Tables D.3, D.4 and the hyperparameters for baselines are in Table D.5. We used the Adam (Kingma and Ba, 2014) optimizer for all training runs. Specifically, Table D.3 contains hyperparameters for program learning. Our use of the hyperparameters during the program learning process are the same as those from Chapter4. Table D.4 contains the hyperparam-

Model	**CalMS21**	**Basketball**
TVAE	1120	895
TVAE+KMeans loss	1079	893
JointVAE	1090	902
VQ-TVAE	971	911
Beta-TVAE	1110	898
Ours (1 program)	1075	894
Ours (2 programs)	1073	893
Ours (3 programs)	1079	899

Table D.2: Median ELBO of CalMS21 and Basketball across 3 runs.

eters for training the VAE component of our model, including the hyperparameters we used for capacity.

	n. epochs	s. epochs	frontier size	penalty	max d	lr	batch size
Synthetic	10	10	30	0.01	2	0.0002	32
CalMS21	6	10	8	0.01	5	0.001	256
Basketball	8	8	30	0.01	3	0.002	128

Table D.3: Hyperparameters for program learning. n. epochs and s. epochs represent the number of neural and symbolic epochs respectively, where the neural epoch is for the neural heuristic. d is depth, and lr is the learning rate.

	epochs	z dim	h dim	RNN dim	adv. dim	disc. cap.	cont. cap.	lr
Synthetic	50	4	16	16	8	0.6	-	0.0002
CalMS21	30	8	256	256	8	0.69	10	0.0001
Basketball	20	8	128	128	8	0.6	4	0.02

Table D.4: Hyperparameters for VAE training. The batch size is the same as the ones for program learning in Table D.3. lr is the learning rate.

	JointVAE			VQ-TVAE	Beta-TVAE		
	weight	disc. C	cont. C	# embs.	weight	C	C iters
CalMS21	100	0.69	10	4	100	20	10k
Basketball	10	0.6	4	2	10	5	20k

Table D.5: Hyperparameters for baseline models. C is the capacity. On CalMS21, the z dim for all baselines are 32 and trained for 200 epochs.

Baseline Details

TVAE. We use a variation of the VAE where the inputs are trajectory data, called a TVAE (Co-Reyes et al., 2018; Zhan et al., 2020; Sun, Kennedy, et al., 2021).

Here, the neural encoder q_ϕ and decoder p_θ are instantiated with recurrent neural networks (RNN), where $\mathbf{z} \sim q_\phi(\cdot|\mathbf{x})$. In this domain, τ is a trajectory of length T: $\tau = \{\mathbf{s}_1, \ldots, \mathbf{s}_T\}$. The TVAE objective is:

$$\mathcal{L}^{\text{tvae}} = \mathbb{E}_{q_\phi}\left[\sum_{t=1}^{T} -\log(p_\theta(\mathbf{s}_t|\mathbf{s}_{<t}, \mathbf{z}))\right] + D_{KL}(q_\phi(\mathbf{z}|\tau)||p(\mathbf{z})). \tag{D.1}$$

All other baselines are variations of the TVAE, based on variations of VAE studied in recent works.

TVAE + KMeans loss. A few works (Ma et al., 2019; Luxem et al., 2020) have studied adding a loss to the VAE framework to encourage clustering in the latent space, called the K-means loss. Given a data matrix $\mathbf{z} \in \mathbb{R}^{d \times N}$, the K-means objective is:

$$\mathcal{L}^{\text{k-means}} = Tr(\mathbf{z}^T\mathbf{z}) - Tr(\mathbf{A}^T\mathbf{z}^T\mathbf{z}\mathbf{A}), \tag{D.2}$$

where $\mathbf{A} \in \mathbb{R}^{N \times k}$ is called the cluster indicator matrix. We optimize this loss using the implementation in (Luxem et al., 2020), where $\mathbf{A}$ is updated by computing the k-first singular values of $\sqrt{\mathbf{z}^t\mathbf{z}}$. The K-means loss is trained jointly with the TVAE loss (Eq D.1) as one of our baselines.

JointVAE. JointVAE (Dupont, 2018) is a variation of VAE that jointly optimizes discrete ($\mathbf{c}$) and continuous ($\mathbf{z}$) latent variables. The JointVAE objective encourages the KL divergence terms to match capacities C_z and C_c that gradually increases during training. The objective is:

$$\begin{aligned}\mathcal{L}^{\text{jointvae}} = \quad & \mathbb{E}_{q_\phi}[\log p_\theta(\mathbf{x}|\mathbf{z}, \mathbf{c})] \\ & - \gamma|D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z})) - C_z| \\ & - \gamma|D_{KL}(q_\phi(\mathbf{c}|\mathbf{x})||p(\mathbf{c})) - C_c|,\end{aligned} \tag{D.3}$$

where γ is a constant. Since the capacities of the discrete and continuous variables are controlled separately, the model is forced to encode information using both channels. Here, we use the trajectory formulation of JointVAE, where:

$$\log p_\theta(\tau|\mathbf{z}, \mathbf{c}) = \sum_{t=1}^{T} \log p_\theta(\mathbf{s}_t|\mathbf{s}_{<t}, \mathbf{z}, \mathbf{c}). \tag{D.4}$$

VQ-TVAE. VQ-VAE (Oord, Vinyals, and Kavukcuoglu, 2017) combines vector quantization with VAEs. These models produce discrete latent encodings that are

used to index an embedding table (or codebook). $\mathbf{z}_e$, the continuous output of the encoder, is mapped to a discrete encoding based on its nearest neighbor in the codebook, then the indexed encoding $\mathbf{z}_q$ is used as input to the decoder. During training, the model learns the codebook, as well as the assignments. The objective is:

$$\mathcal{L}^{\text{vqvae}} = \log p_\theta(\mathbf{x}|\mathbf{z}_q) + ||\text{sg}[\mathbf{z}_e] - e||_2^2 + \beta||\mathbf{z}_e - \text{sg}[e]||_2^2, \tag{D.5}$$

where e are embeddings from the codebook, and sg represents the stopgradient operator.

Beta-TVAE. Beta-VAEs (Higgins et al., 2016; Burgess et al., 2017) have been shown to learn disentangled representations from the image domain. As originally proposed, an adjustable hyperparameter β is used to weigh the KL term in the VAE objective. We use the version of beta-VAE training objective with gradually increasing capacity C proposed in (Burgess et al., 2017). This objective is:

$$\mathcal{L}^{\text{betavae}} = \mathbb{E}_{q_\phi}[\log p_\theta(\mathbf{x}|\mathbf{z})] - \gamma|D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z})) - C|, \tag{D.6}$$

where γ is a constant. Here, we apply the beta-VAE objective to trajectory data using the factorization shown in Eq 5.6.

D.3 Dataset and DSL Details

Synthetic. We generate trajectories with the following steps:

1. Sample initial position $x_1 \sim \mathcal{N}([10, 10], [1, 1])$.
2. Sample velocity from $v = [v_x, v_y] \sim \mathcal{N}([0, 0], [1, 1])$ such that $0.05 < \|v\|_2 < 0.4$.
3. Sample force in x-direction $c_x \sim$ Bernoulli(0.5) and update $v'_x = v_x + 0.4 \cdot (2c_x - 1)$.
4. Sample force in y-direction $c_y \sim$ Bernoulli(0.5) and update $v'_y = v_y + 0.4 \cdot (2c_y - 1)$.
5. Generate trajectory with $x_{t+1} = x_t + v' + 0.2 \cdot \epsilon_t$, where $\epsilon_t \sim \mathcal{N}(0, 1)$.

v' is fixed for an entire trajectory. (c_x, c_y) defines a label for each trajectory (one of 4). The ground-truth decoder is linear with respect to x, v, c_x, c_y. The DSL for the synthetic dataset includes library functions that threshold the final x and y positions,

used to demonstrate that the ground-truth can be learned and the information can be extracted from the neural latent space (Figure 5.4). Experiments were run locally with an Intel 3.6-GHz i7-7700 CPU with 4 cores and an NVIDIA GTX 1080 Ti GPU with 3584 CUDA cores.

CalMS21. The CalMS21 dataset (Sun, Karigo, et al., 2021) consists of trajectory data from a mouse tracker (Segalin et al., 2020), where each mouse is tracked by seven body keypoints from an overhead camera. The two mice are engaging in social interaction, where an intruder mouse is introduced to the cage of the resident mouse. The dataset contains an unlabelled split which we use for training and validation, and we use the test split of Task 1 in CalMS21 for testing. Each frame of the test split is annotated by a domain expert with one of four labels: attack, mount, investigation, and other. We use these annotated behavior labels for comparison with clusters produced by our algorithm. This dataset is available under the CC-BY-NC-SA license.

The feature selects in the CalMS21 DSL are based on behavior attributes computed on trajectory data from domain experts in this area (Segalin et al., 2020). In particular, we asked three domain experts to independently select features from (Segalin et al., 2020) to be part of the DSL. The time it takes domain experts to do this step is on the timescale of minutes. A full list of all features use in the DSLs are as follows:

- Features in DSL 1: head body angle (resident and intruder), social angle (resident and intruder), speed (resident and intruder), distance between nose of resident and tail of intruder, and distance between nose of resident and nose of intruder.
- Features in DSL 2: distance between head of mice, distance between body of mice, distance between head of resident to body of intruder, resident acceleration, resident nose speed, resident axis ratio of fitted ellipse, intersection over union of mice bounding boxes, resident social angle, distance between nose of resident and tail of intruder, and distance between nose of resident and nose of intruder.
- Features in DSL 3: head body angle (resident and intruder), area of ellipse fitted to body keypoints (resident and intruder), acceleration (resident and

intruder), distance between nose of resident and tail of intruder, and distance between nose of resident and nose of intruder.

Note that unless otherwise stated, the CalMS21 experiments uses the features from DSL 1.

The experiments are ran on Amazon EC2 with an Intel 2.3 GHz Xeon CPU with 4 cores equipped with a NVIDIA Tesla M60 GPUs with 2048 CUDA cores.

Basketball. The basketball dataset was also used in (Shah et al., 2020; Zhan et al., 2020) and tracks the xy-positions of players from real NBA games. The positions are centered on the left half-court. Both (5) offensive and (5) defensive players are tracked, as well as the ball (excluded in our experiments).

The DSL for basketball contains library functions that compute the speed, acceleration, final positions, and distance-to-basket of players and take the maximum, minimum, or average over the players. We did not consult a domain expert for this DSL, but these functions were used as labeling functions in (Zhan et al., 2020). Basketball experiments were run locally with an Intel 3.6-GHz i7-7700 CPU with 4 cores and an NVIDIA GTX 1080 Ti GPU with 3584 CUDA cores.

References

Burgess, Christopher P. et al. (2017). "Understanding disentangling in β-VAE". In: *Neural Information Processing Systems Disentanglement Workshop*.

Dupont, Emilien (2018). "Learning disentangled joint continuous and discrete representations". In: *Proceedings of the 32nd Conference on Neural Information Processing Systems*.

Higgins, Irina et al. (2016). "beta-vae: Learning basic visual concepts with a constrained variational framework". In: *International Conference on Learning Representations*.

Kingma, Diederik and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980*.

Luxem, Kevin et al. (2020). "Identifying behavioral structure from deep variational embeddings of animal motion". In: *BioRxiv*.

Ma, Qianli et al. (2019). "Learning representations for time series clustering". In: *Advances in neural information processing systems* 32, pp. 3781–3791.

Oord, Aaron van den, Oriol Vinyals, and Koray Kavukcuoglu (2017). "Neural discrete representation learning". In: *Proceedings of the 31st Conference on Neural Information Processing Systems*.

Co-Reyes, John D et al. (2018). "Self-consistent trajectory autoencoder: Hierarchical reinforcement learning with trajectory embeddings". In: *International Conference on Machine Learning (ICML)*.

Segalin, Cristina et al. (2020). "The Mouse Action Recognition System (MARS): a software pipeline for automated analysis of social behaviors in mice". In: *BioRxiv*.

Shah, Ameesh et al. (2020). "Learning Differentiable Programs with Admissible Neural Heuristics". In: *Advances in Neural Information Processing Systems*. Vol. 33, pp. 4940–4952. URL: https://proceedings.neurips.cc/paper/2020/hash/342285bb2a8cadef22f667eeb6a63732-Abstract.html.

Sun, Jennifer J, Tomomi Karigo, et al. (2021). "The Multi-Agent Behavior Dataset: Mouse Dyadic Social Interactions". In: *arXiv preprint arXiv:2104.02710*.

Sun, Jennifer J, Ann Kennedy, et al. (2021). "Task programming: Learning data efficient behavior representations". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2876–2885. URL: https://openaccess.thecvf.com/content/CVPR2021/html/Sun_Task_Programming_Learning_Data_Efficient_Behavior_Representations_CVPR_2021_paper.html.

Zhan, Eric et al. (2020). "Learning Calibratable Policies using Programmatic Style-Consistency". In: *International Conference on Machine Learning*. PMLR, pp. 11001–11011. URL: https://proceedings.mlr.press/v119/zhan20a.html.