IMITATION LEARNING FOR STYLIZED PHYSICS-BASED CHARACTER CONTROL

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Albert William Tsao

December 2022

© 2022 Albert William Tsao

ALL RIGHTS RESERVED

ABSTRACT

In Computer Graphics, a heavily researched topic is the physical simulation of characters that can exhibit fluid, life-like motions. In recent years, imitation and reinforcement learning techniques have become popular approaches for training such controllers due to their flexibility, generality, and adaptability. One such example is DeepMimic, a data-driven framework that utilizes motion clips along with modern reinforcement learning methods to train control policies for simulated characters that can produce a wide variety of natural notions. Adversarial Motion Priors is an extension upon this framework in which adversarial imitation learning is utilized to enable characters to imitate various motions from a large unstructured dataset of reference motions without the need for explicit synchronization. In this thesis, we adapt the Adversarial Motion Priors framework to be compatible with OpenAI Gym environments. In doing so, a wide variety of RL algorithms can be tested on the framework. Finally, we demonstrate the use of this environment by evaluating Model-based Imitation Learning which is a purely offline imitation learning algorithm that tackles the covariate shift issue common in behavior cloning, a classic offline imitation learning algorithm.

BIOGRAPHICAL SKETCH

Albert Tsao received his undergraduate degree from Cornell University majoring in Computer Science and minoring in Electrical and Computer Engineering. While an undergraduate student, Albert became interested in applications of machine learning that combined Natural Language Processing and Computer Vision.

After finishing his undergraduate studies, he returned to Cornell to join the M.S program. His interests quickly shifted towards reinforcement learning after taking Professor Wen's course on the subject in his first semester. Albert's research applies reinforcement learning in the setting of Computer Graphics and Animation, focusing on imitation learning for simulated characters. He will graduate with a Masters of Science in Computer Science from Cornell University in December 2022. Post-graduation, he plans to spend a few years in industry before potentially pursuing a PhD in reinforcement learning.

To my parents, for their love and unconditional support.

ACKNOWLEDGEMENTS

I'd like to thank my advisor Wen Sun for his support and guidance throughout my M.S. His class on reinforcement learning is what sparked my interest in the field. He's been a great mentor throughout my time here at Cornell and he's taught me much about reinforcement learning and research in general. I would also like to thank my friends and family for their support throughout my time at Cornell as an undergraduate student and M.S. student.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Table of Contents . . . vi
List of Figures . . . viii

1 Introduction 1
1.1 Machine Learning . . . 4
1.2 Reinforcement Learning . . . 5
1.2.1 Markov Decision Processes and Policy . . . 6
1.2.2 Value Function, Q-Function, and Advantage Function . . . 9
1.2.3 TD(λ) for Estimating Value-Function . . . 10
1.2.4 GAE(γ, λ) for Estimating Advantage . . . 11

2 Policy Gradient Algorithms 13
2.1 Importance Sampling . . . 13
2.2 REINFORCE and Advantage-based Policy Gradient . . . 14
2.3 Natural Policy Gradient and Trusted Region Policy Optimization 15
2.4 Proximal Policy Optimization . . . 20

3 Background on Imitation Learning 22
3.1 Behavior Cloning . . . 23
3.2 Generative Adversarial Imitation Learning . . . 24
3.2.1 Maximum Entropy Inverse Reinforcement Learning . . . 24
3.2.2 Generative Adversarial Networks . . . 25
3.2.3 GAIL . . . 26

4 Background on Model-based Imitation Learning 30
4.1 Motivation . . . 30
4.2 MILO Algorithm . . . 31
4.2.1 Datasets . . . 31
4.2.2 Dynamics Model . . . 32
4.2.3 Pessimistic Penalty Design . . . 32
4.2.4 Pessimistic Model-Based Min-Max Imitation Learning . . . 32
4.2.5 Cost Function . . . 34
4.2.6 Updating the Policy . . . 35
4.3 Experiments . . . 36

5 Background on DeepMimic and AMP 38
5.1 DeepMimic . . . 38
5.1.1 The Characters . . . 38
5.1.2 Simulating the World . . . 40
5.1.3 Framing the RL problem . . . 40

5.1.4 The Learning Algorithm . . . 43
5.1.5 Goals and other extensions . . . 44
5.2 AMP . . . 46
5.2.1 States, Actions, and Goals . . . 47
5.2.2 Reward Function . . . 48
5.2.3 The Learning Algorithm . . . 51

6 Developing the Framework and Experimentation 53
6.1 Setting up AMP . . . 53
6.1.1 DeepMimicCore . . . 53
6.1.2 SWIG Wrapper . . . 54
6.2 Augmenting DeepMimicCore . . . 54
6.3 Creating the OpenAI Gym Environment for DeepMimicCore . . . 56
6.3.1 Gym Environment . . . 57
6.3.2 DeepMimicGymEnv . . . 57
6.4 Testing MILO . . . 63
6.4.1 Generating the Offline Dataset . . . 63
6.4.2 Training the Dynamics Model . . . 68
6.4.3 Generating the Expert Dataset . . . 70
6.4.4 SimEnv - Dynamics Model OpenAI Gym Environment . . . 71
6.5 MILO Results . . . 75

7 Future Work 78
7.1 Code-based Future Work . . . 78
7.2 Experimental Future Work . . . 79

8 Conclusion 80

Bibliography 81

LIST OF FIGURES

1.1 Illustration of agent interacting in an MDP. 6

3.1 Image of ALVINN. 23

4.1 This figure is taken directly from the MILO paper experiments section (figure 2). It shows learning curves across five seeds for MILO against BC after 1000 epochs of training and ValueDICE after 10 thousand iterations. The normalized graph shows the normalized scores for the expert. 37

5.1 Figures of the four characters in the DeepMimic and AMP framework. 39

6.1 Figure 6.1a shows the character being reset to t = 0 and matching the reference motion like in the original AMP framework. Figure 6.1b resets to t = 0 but each joint (except the hip and ankle) has a random angle added to it. The angle of each added rotation is sampled from $\mathcal{U}(-40, 40)$ degrees and has no knee rotation. Figure 6.1c follows the same procedure as Figure 6.1b but has knee rotation. Figure 6.1d resets to t = 0 but instead adds noise by adding a random noise vector $\mathcal{U}(-0.005, 0.005)$ to the entire state. 55

6.2 Character is reset to t = 0 but a vector of random noise is added to pose and velocity. The noise is sampled from $\mathcal{U}(-0.1, 0.1)$. As we can see, the character becomes disfigured due to this added noise. 56

6.3 Average statistics for various checkpoints collected from training a spinkick character. 65

6.4 The DTW Costs and lengths of trajectories in D_1 and D_2 66

6.5 Behavior Cloning results for various datasets. 67

6.6 Dynamic Model Training and Validation Loss. 69

6.7 The above graphs show the error in the dynamics model when replaying trajectories. The error at each timestep is computed using the ℓ_2 norm between the current true state s_t and the state in the trajectory rolled out in the dynamics model, $\hat{s}_t$. Figure 6.7a shows the average absolute error $\|s_t - \hat{s}_t\|_2$ at each timestep and Figure 6.7b shows the average relative error $\frac{\|s_t - \hat{s}_t\|_2}{\|s_t\|_2}$ at each timestep. 70

6.8 Figure 6.8a shows the average IPM return which is defined as the sum of the IPM rewards, $-f(s, s')$, for a trajectory. Figure 6.8b shows the lengths of the trajectories inside the model. 76

6.9 Figure 6.9a shows the average DTW cost for trajectories sampled in the real environment while Figure 6.9b shows the average length of those sampled trajectories. 77

CHAPTER 1

INTRODUCTION

In recent years, physically simulated characters have seen increased usage due to a rise in popularity of animated movies, video games, and new technologies such as virtual and augmented reality. This has led to an increased demand for systems that can train characters to exhibit a wide-range of human-like behaviors in a variety of situations.

Traditional techniques for simulating such characters include kinematic and physics-based methods. Kinematic methods are not concerned with the actual physics that decides the motion of the character, but rather with describing the motion itself. They typically rely on a large dataset of motion clips [14] to train controllers that select which clips to use in a given situation. Data-driven methods such as Gaussian processes [15] can learn latent representations to synthesize motions, while more recently, machine learning models such as autoencoders can be used to create generative models. While kinematic methods can produce characters that move realistically in many different situations, they often struggle to synthesize accurate real-life behavior for novel situations where there isn't much data.

Rather than relying on motion data to create controllers, physics-based methods utilize motion equations or physics simulations to generate the motions [23]. Since these motions are derived from first principles, the physics-based controllers create realistic motions for characters even in novel situations. The downside is the need for human insight to design heuristics or objectives for the controller that lead to natural behaviors. Common heuristics include symmetric, stability, etc., but can vary greatly on the task and character itself.

Reinforcement learning (RL) has seen increased usage for developing controllers in the past decade. In RL, agents interact with the environment while relying on a reward signal to learn. Prior works have used deep neural networks to train controllers to produce complex behaviors. However, these controllers often exhibit artifacts such as unusual body motions or unrealistic postures. DeepMimic [19] addresses these issues by directly rewarding the agent for producing motions like those in the reference motion data. Specifically, the reward at a given time is based on the difference in pose between the character and the target pose in the reference motion. In order to choose the correct target pose, a phase variable is used to synchronize the character with the reference motion. While this strategy is very effective at imitating single motion clips, it does not scale well for training controllers on datasets that contain various motions. In this case, the goal is to use the various motions to train a controller that can exhibit motions that match the "style" defined by the dataset without needing to explicitly match any single motion. DeepMimic and similar methods that use a phase variable for synchronization struggle with such datasets as it is generally not possible to synchronize multiple reference motions with a phase variable.

Adversarial Motion Priors (AMP) [20] is an extension of DeepMimic by the same author that attempts to address this issue using imitation learning. In imitation learning, the reward function is not known as in the RL setting, so it often needs to be learned. AMP uses adversarial imitation learning to learn a reward function that gives a reward based on how close the motions of the character mimic the style of the motions in the dataset. Rather than trying to synchronize the character to the reference motion like in DeepMimic, AMP trains the controller to synthesize motions similar in style to the reference motions.

AMP uses imitation learning while having the agent interact with the environment. There is a broad class of imitation learning algorithms that fall into the offline setting in which the agent does not have access to the environment but instead learns from an expert dataset. Offline imitation learning algorithms often suffer from the issue of covariate shift. At a high-level, this is an issue that can cause the agent to perform poorly in the environment due to the difference in distribution in the training dataset and the data seen when the agent interacts with the environment. Model-based Imitation Learning from Offline Data (MILO) [5] is an offline imitation learning algorithm that tackles the issue of covariate shift in both theory and practice. In this thesis, we will be primarily testing how MILO performs in the AMP framework.

In this thesis, we first give background knowledge on reinforcement learning, imitation learning, and the relevant algorithms used in our research. This background knowledge is necessary for understanding MILO in chapter 4 as well as the algorithms used in the DeepMimic and AMP frameworks discussed in chapter 5. Chapter 6 describes the main research done for the thesis. Specifically, we augmented the AMP framework in order to make it compatible with OpenAI Gym [4] environments. OpenAI Gym environments all follow a standard API, making it very easy to test various RL algorithms on various environments and benchmarks. Once we created the gym environment for AMP, we tested MILO on the AMP framework and illustrated the process involved in data gathering and model training as well as the results. Finally, chapter 7 discusses future work.

1.1 Machine Learning

At a high level, machine learning is a field of study that aims to develop and understand algorithms that enable computers to improve with experience. These algorithms extract knowledge from training data using various methods to train models that can make predictions without having a codified ruleset. Over the past few decades, machine learning has exploded in popularity, especially with the rise of deep learning. From recommendation systems to spam detection to language translation, machine learning has been applied to countless domains. Advancements in architectures such as Transformers [38] in Natural Language Processing and convolutions neural networks [13] in Computer Vision have allowed for rapid developments of consumer technologies such as voice-assistant tools and image-recognition systems.

Machine learning is typically categorized into three categories: supervised learning, unsupervised learning, and reinforcement learning. The main difference between the three is how models are trained and the type of feedback available. Supervised learning, the most common type of machine learning used in practice, uses labeled training data to learn a mathematical model that maps inputs to the desired outputs. Common algorithms include Naive Bayes, Linear Regression, SVMs, and Neural Networks. In contrast to supervised learning, unsupervised learning attempt to find patterns and structure in unlabeled data. Neural networks such as Variational Autoencoders [12] are often used for this task. However, probabilistic methods based on clustering or principal component analysis are also common. There exists a category of machine learning algorithms that falls in between supervised and unsupervised learning: semi-supervised learning. As the name implies, semi-supervised learning

concerns itself with learning from a small sample of labeled example combined with a large amount of unlabeled data. This approach to machine learning has become popular in areas such as Computer Vision, where the hope is to leverage the massive amounts of images publicly available for training. Many semi-supervised algorithms use some sort of consistency regularization or pseudo-labeling such as FixMatch [30] in Computer Vision.

Finally, the third large category of machine learning is reinforcement learning. Due to reinforcement learning being heavily used in our experiments, the following section has been dedicated to giving the relevant background on the topic.

1.2 Reinforcement Learning

Reinforcement learning (RL) is broadly concerned with training a model to learn how to act in an environment by maximizing total reward (or minimizing total cost). The feedback to the action's agent is given in the form of a numerical reward (or cost). By exploring the environment, the hope is that the learner can find the optimal actions to take. However, characteristics of the environment, such as having a delayed reward signal or having a partial observable state can make this task very challenging.

RL varies greatly from the paradigms of supervised and unsupervised learning. While supervised learning focuses solely on training using a labeled dataset, it is not fit for problems that require interaction with the environment. It is often the case in RL that the agent must explore the environment and gather its own data. Additionally, RL is not concerned with finding hidden structures

in data like unsupervised learning. The following sections give a brief summary of the background knowledge required to understand the algorithms used in our research

1.2.1 Markov Decision Processes and Policy

MDP

Markov decision processes (MDPs) serve as a mathematical framework for the learning problem. The learner is called the agent which interacts with the environment as described in Figure 1.1 below.

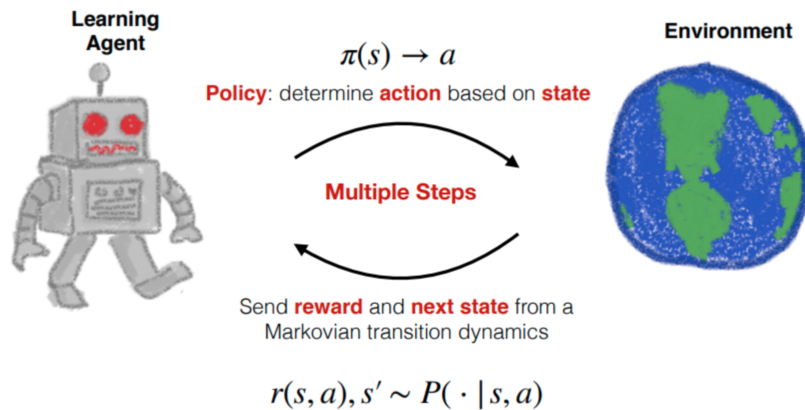


Figure 1.1: Illustration of agent interacting in an MDP.

For the purposes of this thesis, the emphasis will be placed on the stationary, finite-horizon MDP. In theoretical analysis, finite-horizon MDPs typically have a time dependence, meaning the policies, rewards, and utility functions (e.g., value function, Q-function, and advantage function) vary with time. However, the discussion here will focus on a stationary MDP. Additionally, this MDP will

not use a discount factor, γ, which is typical of finite horizon MDPs. However, we will see that certain algorithms will use a discount factor in computations despite being the horizon setting. MDPs are typically specified in the following manner:

$$\mathcal{M} = \{\mathcal{S}, \mathcal{A}, P, r, \mu, T\}$$

- $\mathcal{S}$ represents the state space which may be finite or infinite.
- $\mathcal{A}$ represents the action space which may be finite or infinite.
- $P : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ represents the dynamics of our MDP. If the agent takes action a in state s, $P(s'|s, a)$ represents the probability of transitioning to state s'.
- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ represents the reward function. $r(s, a)$ is the reward the agent receives upon taking action a in state s. In some cases, the MDP specifies a cost function $c : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ instead of a reward function. In this case, the goal of the agent is to learn how to interact in the environment in order to minimize total cost.
- μ represents the initial state distribution $\mu \in \Delta(\mathcal{S})$. Whenever our agent resets after finishing a trajectory or episode, the initial state is sampled from μ. In certain circumstances, the initial state is fixed at s_0.
- T represents the horizon length. The horizon is maximum possible length of a trajectory or sequence of states and actions $\{s_0, a_0, \cdots, s_{T-1}, a_{T-1}\}$ until termination. Additionally, we often store an additional last state as the termination state s_T.

Policy

The policy is the agent which interacts with the environment and is typically represented by π. In this thesis, we will consider stationary policies $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ in which the agent decides an action only based on the current state. However, in general, the policy can decide its action from the entire history of observations. $\pi(s)$ will denote the action taken in state s while $\pi(a|s)$ represents the probability π decides on action a in state s.

Trajectories

MDP's extend upon Markov chains by adding the notion of actions and rewards. The transition dynamics use the Markov property as the transition probability is only influenced by the current state and action. With the Markov property, the probability of seeing the trajectory $\tau = \{s_0, a_0, \cdots, s_t, a_t\}$, following π, can be written as:

$$\mathbb{P}^{\pi}(s_0, a_0, \cdots, s_t, a_t) = \mu(s_0)\pi(a_0|s_0)P(s_1|s_0, a_0)\pi(a_1|s_1)\cdots P(s_t|s_{t-1}, a_{t-1})\pi(a_t|s_t)$$

The probability of π visiting state (s, a) at time-step t can be written by marginalizing over previous t steps and actions $\{s_0, a_0, \cdots, s_{t-1}, a_{t-1}\}$:

$$\mathbb{P}^{\pi}_t(s, a; \mu) = \sum_{s_0, a_0, \cdots, s_{t-1}, a_{t-1}} \mathbb{P}^{\pi}(s_0, a_0, \cdots, s_{t-1}, a_{t-1}, s_t = s, a_t = a)$$

Finally, the average state-action distribution (in finite MDPs) can be written as:

$$d^{\pi}(s, a) = \frac{1}{T} \sum_{t=0}^{T-1} \mathbb{P}^{\pi}_t(s, a; \mu)$$

Alternatively, we can define the average state-action distribution using time-indexing:

$$d_0^\pi = \mu \qquad d_t^\pi(s,a) = \sum_{s',a'} d_{t-1}^\pi(s',a')P(s|s',a')\pi(a|s) \qquad d^\pi(s,a) = \frac{1}{T}\sum_{t=0}^{t-1} d_t^\pi(s,a)$$

The state-distribution is simply the state-action distribution marginalized over the current action:

$$d^\pi(s) = \sum_a d^\pi(s,a)$$

1.2.2 Value Function, Q-Function, and Advantage Function

These three functions give different measures of utility for the policy. The value function in a stationary, finite-horizon MDP is defined as:

$$V^\pi(s) = \mathbb{E}[\sum_{t=0}^{T-1} r(s_t,a_t)|\pi, s_0 = s] \tag{1.1}$$

Using d^π, $V^\pi(s)$ can also be written as $T \underset{(s,a)\sim d^\pi}{\mathbb{E}}[r(s,a)]$. The value-function is the expected return, or cumulative reward, for policy π when starting in state s. A similar function is the Q-function:

$$Q^\pi(s,a) = \mathbb{E}[\sum_{t=0}^{T-1} r(s_t,a_t)|\pi, s_0 = s, a_0 = a] \tag{1.2}$$

In the Q-function, the policy π starts in state s and takes action a which may be different than $\pi(s)$. It is clear to see that $V^\pi(s) = Q^\pi(s,\pi(s))$ The value function and Q-function naturally lead to the advantage function:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{1.3}$$

$A^\pi(s,a)$ indicates how much better (or worse) taking action a in state s is compared to $\pi(s)$.

1.2.3 TD(λ) for Estimating Value-Function

In small MDPs, the value-function can be represented as a lookup table. However, in most real-world scenarios, the state and action space are extremely large, even infinite. Computing the value-function exactly not only takes too much computational time but storing the function as a table becomes intractable due to memory constraints. Thus, value-functions are instead estimated using a neural network, V_θ, parameterized by θ.

TD(λ) [34] is an algorithm that can be used to train V_θ by using gradient descent. Assuming we roll out trajectories and sample random states to form the batch $\{s^i\}_{i=1}^N$, the optimization problem is:

$$\min_\theta \frac{1}{2N} \sum_{i=1}^{N} (y^i - V_\theta(s^i))^2 \tag{1.4}$$

where s^i represents the ith sample state in our batch and y^i is the target for $V_\theta(s^i)$. One way to compute the target y for state s is to simply compute the Monte-Carlo return. In order to compute the target for a state s, we introduce the concept of $n-$step returns. Consider a trajectory (or episode in Monte Carlo) $\tau = \{s_0, a_0, \cdots, s_{T-1}, a_{T-1}\}$. The n-step return starting from state s_t with discount factor γ can be defined as:

$$R_t^n(\tau, \gamma) = \sum_{\ell=0}^{n-1} \gamma^\ell r(s_{t+\ell}, a_{t+\ell}) + \gamma^n V_\theta(s_{t+n})$$

where the assumption is that any rewards past the horizon $T - 1$ are zero. That is, $r(s_{t+\ell}, a_{t+\ell}) = 0$ for $\ell \geqslant T - t$ which means $R_t^n(\tau, \gamma) = R_t^{T-t}(\tau, \gamma)$ for $n \geqslant T - t$. If we were to compute $R_t^\infty(\tau, \gamma)$, this would give the original Monte-Carlo return $\sum_{\ell=0}^\infty \gamma^\ell r(s_{t+\ell}, a_{t+\ell})$ which we could use for the target. While this is an unbiased estimate, this has a high variance. TD(λ) uses a biased but low variance estimator to generate the target $V_\theta(s)$ by computing an exponentially-weighted

average of the $n-$step returns using decay parameter λ:

$$R_t(\tau, \gamma, \lambda) = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^n(\tau, \gamma) \tag{1.5}$$

As mentioned earlier, since $R_t^n(\tau, \gamma) = R_t^{T-t}(\tau, \gamma)$ for $n \geqslant T - t$, we can write

$$R_t(\tau, \gamma, \lambda) = (1 - \lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^n(\tau, \gamma) + \lambda^{T-t-1} R_t^{T-t}(\tau, \gamma)$$

The update rule at iteration k with step size α is:

$$\begin{aligned} \theta_{k+1} &= \theta_k - \frac{1}{2N} \alpha \sum_{i=1}^{N} \nabla_\theta [y^i - V_{\theta_k}(s^i)]^2 \\ &= \theta_k + \frac{\alpha}{N} \sum_{i=1}^{N} [y^i - V_{\theta_k}(s^i)] \nabla_\theta V_{\theta_k}(s^i) \end{aligned} \tag{1.6}$$

If we have a set of trajectories $D = \{\tau_i\}$ where $\tau_i = \{s_t^i, a_t^i, r_t^i\}_{t=0}^{T-1}$ and s_t^i represents the tth state in trajectory i, then the update rule is:

$$\begin{aligned} \theta_{k+1} &= \theta_k - \frac{1}{2|D|T} \alpha \sum_{i=1}^{|D|} \sum_{t=0}^{T-1} \nabla_\theta [y_t^i - V_{\theta_k}(s_t^i)]^2 \\ &= \theta_k + \frac{\alpha}{|D|T} \sum_{i=1}^{|D|} \sum_{t=0}^{T-1} [y_t^i - V_{\theta_k}(s_t^i)] \nabla_\theta V_{\theta_k}(s_t^i) \end{aligned} \tag{1.7}$$

1.2.4 GAE(γ, λ) for Estimating Advantage

Estimates of the advantage function are often needed, especially in the policy gradient algorithms detailed in chapter 2. Generalized Advantage Estimation (GAE) [28], is one such algorithm that makes use of ideas from TD(λ). Given a trajectory τ, define the TD residual of Value function V with discount factor γ at timestep t as:

$$\delta_t^V(\tau) = r(s_t, a_t) + \gamma V(s_{t+1}) - V(s_t)$$

Then, define the sum of k δ terms as:

$$\hat{A}_t^{(k)}(\tau, \gamma) = \sum_{\ell=0}^{k-1} \gamma^\ell \delta_{t+\ell}^V(\tau) = \sum_{\ell=0}^{k-1} \gamma^\ell r(s_{t+\ell}, a_{t+\ell}) + \gamma^k V(s_{t+k}) - V(s_t)$$

where any rewards past the horizon $T-1$ are zero. GAE(γ, λ), is the exponentially-weighted average of the k-step estimators

$$\begin{aligned} \hat{A}_t^{GAE(\gamma,\lambda)}(\tau) &= (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} \hat{A}_t^{(n)}(\tau) \\ &= \sum_{\ell=0}^{\infty} (\gamma\lambda)^\ell \delta_{t+\ell}^V \end{aligned} \tag{1.8}$$

GAE(γ, λ) can also be written in terms of the TD(λ) return as

$$\hat{A}_t^{GAE(\gamma,\lambda)}(\tau) = R_t(\tau, \gamma, \lambda) - V(s_t)$$

CHAPTER 2

POLICY GRADIENT ALGORITHMS

In this chapter, we briefly introduce several policy gradient algorithms, specifically in the finite horizon setting $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, P, r, \mu, T\}$. Define $\{\pi_\theta | \theta \in \Theta \subset \mathbb{R}^d\}$ as a class of parametric polices. In policy gradient algorithms, gradient methods are used to maximize expected return. The algorithms presented in the following sections are actor-critic methods in that the actor, the policy, and the critic, the value-function, are both learned.

2.1 Importance Sampling

We first introduce importance sampling, an important mathematical tool for deriving policy gradient algorithms. Importance sampling is a technique for getting an unbiased estimate of an expected value under one distribution using samples from another distribution. This is used in abundance in RL, such as in off-policy policy gradient algorithms where we use samples from a separate policy π^b to estimate the gradient of π.

To form an unbiased estimate $\mathop{\mathbb{E}}_{x \sim A}[f(x)]$ using samples from a separate distribution, B, the expression is first rewritten as:

$$\mathop{\mathbb{E}}_{x \sim A}[f(x)] = \mathop{\mathbb{E}}_{x \sim B} \frac{A(x)}{B(x)} f(x)$$

assuming that $\max_x \frac{A(x)}{B(x)} < \infty$. Then, an unbiased estimate is simply $\frac{1}{N} \sum_{i=1}^{N} \frac{A(x)}{B(x)} f(x)$ where the samples are drawn from B.

2.2 REINFORCE and Advantage-based Policy Gradient

REINFORCE [33] is one of the classic policy gradient algorithms in reinforcement learning. Define the discounted total reward of a trajectory in the finite-horizon setting as:

$$R(\tau) := \sum_{t=0}^{T-1} r(s_h, a_h)$$

Additionally, define $\rho_\theta(\tau)$ as the distribution from which trajectories $\tau = \{s_0, a_0, \cdots, s_{T-1}, a_{T-1}\}$ are sampled:

$$\rho_\theta(\tau) = \mu(s_0)\pi_\theta(a_0|s_0)P(s_1|s_0, a_0)\pi_\theta(a_1|s_1)\cdots P(s_{T-1}|s_{T-2}, a_{T-2})\pi(a_{T-1}|s_{T-1})$$

The objective is to maximize $J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}[R(\tau)]$. Using importance weighting, the gradient for the objective function in REINFORCE is:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\left(\sum_{t=0}^{T-1}\nabla_\theta \ln \pi_\theta(a_t|s_t)\right)R(\tau)\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T-1}\nabla_\theta \ln \pi_\theta(a_t|s_t)\left(\sum_{\ell=0}^{T-1} r(s_\ell, a_\ell)\right)\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T-1}\nabla_\theta \ln \pi_\theta(a_t|s_t)\left(\sum_{\ell=t}^{T-1} r(s_\ell, a_\ell)\right)\right] \qquad (2.1)
\end{aligned}$$

The last equality comes from the fact that $\mathop{\mathbb{E}}_{\tau\sim\rho_\theta}[\nabla_\theta \ln \pi_\theta(a_t|s_t) r_\ell] = 0$ for any ℓ, t, with $\ell < t$. It can also be shown that for any state-dependent baseline, $b(s)$,

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T-1}\nabla_\theta \ln \pi_\theta(a_t|s_t)\left((\sum_{\ell=0}^{T-1} r(s_\ell, a_\ell)) - b(s)\right)\right]$$

If $b(s) = V^{\pi_\theta}(s)$, this naturally leads to an advantage-based version of the gradient which has several forms:

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathop{\mathbb{E}}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T-1}\nabla_\theta \ln \pi_\theta(a_t|s_t)A^{\pi_\theta}(s_t,a_t)\right] \\ &= \sum_{t=0}^{T-1}\mathop{\mathbb{E}}_{s,a\sim\mathbb{P}_t^{\pi_\theta}}\left[\nabla_\theta \ln \pi_\theta(a|s)A^{\pi_\theta}(s,a)\right] \\ &= T\mathop{\mathbb{E}}_{s\sim d^{\pi_\theta}}\mathop{\mathbb{E}}_{a\sim\pi_\theta(\cdot|s)}\left[\nabla_\theta \ln \pi_\theta(a|s)A^{\pi_\theta}(s,a)\right]\end{aligned} \tag{2.2}$$

Finally, the update rule for REINFORCE at iteration k is:

$$\theta_{k+1} = \theta_k + \alpha\nabla_\theta \widehat{J}(\theta_k) \tag{2.3}$$

where α is the step size and $\nabla_\theta \widehat{J}(\theta_k)$ is an unbiased estimate of $\nabla_\theta J(\theta_k)$

2.3 Natural Policy Gradient and Trusted Region Policy Optimization

At a high level, Natural Policy Gradient (NPG) [10] and Trusted Region Policy Optimization (TRPO) [27] are two policy gradient methods that choose their descent direction based on the local geometry of the manifold induced by the parameters of the policy. Additionally, they ensure the policy update is not too large by constraining the policy to stay within some trust region defined using the Kullback-Leibler (KL) divergence. Both algorithms are extremely similar with TRPO giving a more generalized algorithm that can be tied back to NPG.

Both algorithms can be derived by starting with the trust region formulation for policy updates [31]. At iteration k of the algorithm, the objective function

with constraints is:

$$\begin{aligned} \underset{\theta}{\text{maximize}} \quad & \mathop{\mathbb{E}}_{s \sim d^{\pi_{\theta_k}}} \mathop{\mathbb{E}}_{a \sim \pi_{\theta_k}(\cdot|s)} \left[A^{\pi_{\theta_k}}(s,a) \right] \\ \text{subject to} \quad & KL(\rho_{\pi_{\theta_k}} | \rho_{\pi_\theta}) \leqslant \delta \end{aligned} \tag{2.4}$$

where δ is a hyperparameter. Given two distributions $P \in \Delta(X)$ and $Q \in \Delta(X)$, the KL-divergence KL($P|Q$) is:

$$KL(P|Q) = \mathop{\mathbb{E}}_{x \sim P} \left[\ln \frac{P(x)}{Q(x)} \right]$$

In TRPO, the objective is written in a slightly different but equivalent form using importance weighting:

$$\mathop{\mathbb{E}}_{s \sim d^{\pi_{\theta_k}}} \mathop{\mathbb{E}}_{a \sim \pi_{\theta_k}(\cdot|s)} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a) \right]$$

In either case, the update rule says to update the policy to maximize local advantage while staying within the trust region defined by the KL-divergence. As written, it is unclear how to optimize the above problem, so a first-order Taylor expansion is performed on the objective at θ_k. The objective becomes:

$$\max_\theta \nabla_\theta J(\theta_k)^\top (\theta - \theta_k)$$

where $\nabla_\theta J(\theta_k)$ is the advantage-based version of the gradient shown in the previous section. Since the KL-divergence is not symmetric, it does not serve as a valid distance metric, so a second-order Taylor expansion of the constraint at θ_k. After performing the second-order Taylor expansion, the new constraint becomes:

$$(\theta - \theta_k)^\top F_{\theta_k} (\theta - \theta_k) \leqslant 2\delta$$

where F_{θ_k} is defined as the Fisher information matrix:

$$F_{\theta_k} = T \mathop{\mathbb{E}}_{s,a \sim d^{\pi_{\theta_k}}} (\nabla \ln \pi_{\theta_k}(a|s))(\nabla \ln \pi_{\theta_k}(a|s))^\top \tag{2.5}$$

It is clear to see that the Fisher information matrix is a symmetric positive semi-definite matrix . The derivation of the simplified constraint uses the following facts about the gradient and Hessian of the KL-divergence:

$$\nabla KL(\rho_{\pi_{\theta_k}} | \rho_{\pi_\theta})|_{\theta=\theta_k} = 0$$
$$\nabla^2 KL(\rho_{\pi_{\theta_k}} | \rho_{\pi_\theta})|_{\theta=\theta_k} = T \mathop{\mathbb{E}}_{s,a\sim d^{\pi_{\theta_k}}} (\nabla \ln \pi_{\theta_k}(a|s))(\nabla \ln \pi_{\theta_k}(a|s))^\top$$

The final optimization problem at iteration k becomes:

$$\begin{aligned} \underset{\theta}{\text{maximize}} \quad & \nabla_\theta J(\theta_k)^\top (\theta - \theta_k) \\ \text{subject to} \quad & (\theta - \theta_k)^\top F_{\theta_k} (\theta - \theta_k) \leqslant 2\delta \end{aligned} \tag{2.6}$$

This maximization problem can be solved in close form by solving:

$$\begin{aligned} \underset{v}{\text{maximize}} \quad & c^\top v \\ \text{subject to} \quad & \|v\|_2^2 \leqslant 2\delta \\ \text{where} \quad & v = F_{\theta_k}^{\frac{1}{2}} (\theta - \theta_k), c = F_{\theta_k}^{-\frac{1}{2}} \nabla_\theta J(\theta_k) \end{aligned}$$

Solving this simple constrained optimization problem exactly leads to the following update at iteration k:

$$\theta_{k+1} = \theta_k + \alpha F_{\theta_k}^{-1} \nabla_\theta J(\theta_k) \tag{2.7}$$

where α, the step size, is defined as:

$$\alpha = \sqrt{\frac{2\delta}{(\nabla_\theta J(\theta_k))^\top F_{\theta_k}^{-1} \nabla_\theta J(\theta_k)}}$$

The NPG algorithm stops here with this exact update rule. However, due to the approximation done with the first and second order Taylor expansions, the original KL constraint may be violated. Thus, TRPO goes one step further and uses a line search to find the best step size to ensure that the objective is improved while making sure the KL-constraint is satisfied.

The procedure for running NGP and TRPO is shown in Algorithm 2.1 below. Since explicitly computing the matrix inverse $F_{\theta_k}^{-1}$ is too computationally expensive, the conjugate gradient algorithm is used to solve $F_{\theta_k} x = \nabla_\theta J(\theta_k)$ for $x = F_{\theta_k}^{-1} \nabla_\theta J(\theta_k)$. Automatic differentiation systems, such as those in PyTorch [18], can be used to compute the Hessian-vector product, Hx, needed in conjugate gradient.

Algorithm 2.1 NPG and TRPO

1: **Input:** policy parameters θ_0, value function parameters ψ_0, KL-divergence parameter δ, **TRPO only**: backtracking coefficient α and max depth of backtracking K
2: **for** $k = 0, 1, 2, \cdots$ **do**
3: Collect set of trajectories $D_k = \{\tau^i\}$ where $\tau^i = \{s_t^i, a_t^i, r_t^i\}_{t=0}^{T-1}$ by rolling out using $\pi_k = \pi(\theta_k)$ in the environment and collect rewards.
4: Compute advantage estimates, $\hat{A}(s,a)$, of $A^{\pi_{\theta_k}}(s,a)$ using any advantage estimate algorithm based on V_{ψ_k}.
5: Form the policy gradient estimate:

$$\nabla_\theta \widehat{J}(\theta_k) = \frac{1}{|D_k|} \sum_{i=0}^{|D_k|-1} \left[\sum_{t=0}^{T-1} \nabla_\theta \ln \pi_{\theta_k}(a_t^i | s_t^i) \hat{A}(s_t^i, a_t^i) \right]$$

6: Form the Fisher information matrix estimate:

$$\widehat{F_{\theta_k}} = \frac{1}{|D_k|} \sum_{i=0}^{|D|-1} \left[\sum_{t=0}^{T-1} \nabla_\theta \ln \pi_{\theta_k}(a_t^i | s_t^i) \nabla_\theta \ln \pi_{\theta_k}(a_t^i | s_t^i)^\top \right]$$

7: Use conjugate gradient algorithm to compute $\widehat{F}_{\theta_k}^{-1} \nabla_\theta \widehat{J}(\theta_k)$.
8: **if** NPG **then**
9: Update the policy with the NPG update rule:

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{(\nabla_\theta \widehat{J}(\theta_k))^\top \widehat{F}_{\theta_k}^{-1} \nabla_\theta \widehat{J}(\theta_k)}} \widehat{F}_{\theta_k}^{-1} \nabla_\theta \widehat{J}(\theta_k)$$

10: **else**
11: Update the policy with the TRPO update rule:

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{(\nabla_\theta \widehat{J}(\theta_k))^\top \widehat{F}_{\theta_k}^{-1} \nabla_\theta \widehat{J}(\theta_k)}} \widehat{F}_{\theta_k}^{-1} \nabla_\theta \widehat{J}(\theta_k)$$

where $j \in \{0, \cdots, K\}$ is the smallest value found after line search that ensures the objective is improved while ensuring the KL-divergence constraint is satisfied.
12: **end if**
13: Fit value-function using regression on MSE loss:

$$\psi_{k+1} = \underset{\psi}{\operatorname{argmin}} \frac{1}{2|D_k|T} \sum_{i=0}^{|D_k|-1} \sum_{t=0}^{T-1} (y^i - V_\theta(s_t^i))^2$$

14: **end for**

2.4 Proximal Policy Optimization

Proximal Policy Optimization (PPO) [29], like TRPO and NPG, is an algorithm that also aims to improve the policy without making such a large change that would cause policy degradation. While TRPO and NPG are second-order methods that rely on the Hessian Fisher information matrix, PPO is a first-order method that only relies on computing the gradient. There are two variants of the PPO algorithms: PPO-Penalty and PPO-Clip. The former penalizes the KL-divergence in the objective instead of making it a hard constraint. The latter algorithm doesn't have a KL-divergence constraint or penalty in the objective. Instead, it uses clipping to avoid policy degradation due to large policy updates. We will focus on PPO-clip in this section.

Rather than using a KL-divergence constraint for the trust region, PPO specifies the trust region based on total variation distance:

$$\begin{aligned} \underset{\theta}{\operatorname{argmax}} \quad & \underset{s \sim d^{\pi_{\theta_k}}}{\mathbb{E}} \underset{a \sim \pi_{\theta_k}(\cdot|s)}{\mathbb{E}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a) \right] \\ \text{subject to} \quad & \sup_s \|\pi_\theta(\cdot|s) - \pi_{\theta_k}(\cdot|s)\|_{tv} \leqslant \delta \end{aligned} \tag{2.8}$$

Note that the objective uses importance weighting. This allows us to sample $d^{\pi_{\theta_k}}$ and π_{θ_k} in order to estimate this expectation. Rather than using this constraint, PPO-Clip approximates it by modifying the objective with specialized clipping:

$$\underset{\theta}{\operatorname{argmax}} \quad \underset{s \sim d^{\pi_{\theta_k}}}{\mathbb{E}} \underset{a \sim \pi_{\theta_k}(\cdot|s)}{\mathbb{E}} \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), \operatorname{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s,a) \right) \tag{2.9}$$

where ϵ is a hyperparameter that controls how far the new policy can be away

from the policy. The clip function here is defined as:

$$\text{clip}(\frac{\pi_\theta}{\pi_{\theta_k}}, 1-\epsilon, 1+\epsilon) = \begin{cases} 1-\epsilon & \text{if } \frac{\pi_\theta}{\pi_{\theta_k}} \leqslant 1-\epsilon \\ 1+\epsilon & \text{if } \frac{\pi_\theta}{\pi_{\theta_k}} \geqslant 1+\epsilon \\ \frac{\pi_\theta}{\pi_{\theta_k}} & \text{otherwise} \end{cases}$$

When $\frac{\pi_\theta}{\pi_{\theta_k}}$ is outside the range $[1-\epsilon, 1+\epsilon]$, the gradient is zero, ensuring that π_{θ_k} doesn't update to a policy far away. The outer min also ensures this as well. When $A^{\pi_{\theta_k}}$ is positive, $\pi_\theta(a|s)$ is increased so more weight is placed on to action a at state s. However, the min and clipping ensures that at maximum, $\pi_\theta(a|s) = (1+\epsilon)\pi_{\theta_k}(a|s)$. That is, π_θ cannot move too far from π_{θ_k}. A similar argument can be made for when $A^{\pi_{\theta_k}}$ is negative, less weight should be placed on $\pi_\theta(a|s)$ but the largest amount π_θ can move is $\pi_\theta(a|s) = (1-\epsilon)\pi_{\theta_k}(a|s)$

The algorithm for PPO is shown below in Algorithm 2.2. A key difference between PPO and NGP/TRPO is that PPO uses multiple steps of mini-batch stochastic gradient ascent in order to optimize the objective function, whereas NPG/TRPO just use the single-step gradient update.

Algorithm 2.2 PPO

1: **Input:** policy parameters θ_0, value function parameters ψ_0
2: **for** $k = 0, 1, 2, \cdots$ **do**
3: Collect set of trajectories $D_k = \{\tau^i\}$ where $\tau^i = \{s_t^i, a_t^i, r_t^i\}_{t=0}^{T-1}$ by rolling out using $\pi_k = \pi(\theta_k)$ in the environment and collect rewards.
4: Compute advantage estimates, $\hat{A}(s,a)$, of $A^{\pi_{\theta_k}}$(s,a) using any advantage estimate algorithm based on V_{ψ_k}.
5: Use mini-batch SGA to maximize the objective:

$$\theta_{k+1} = \underset{\theta}{\text{argmax}} \underset{s \sim d^{\pi_{\theta_k}}}{\mathbb{E}} \underset{a \sim \pi_{\theta_k}(\cdot|s)}{\mathbb{E}} \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), clip\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s,a)\right)$$

6: Fit value-function using regression on MSE loss:

$$\psi_{k+1} = \underset{\psi}{\text{argmin}} \frac{1}{2|D_k|T} \sum_{i=0}^{|D_k|-1} \sum_{t=0}^{T-1} (y^i - V_\theta(s_t^i))^2$$

7: **end for**

CHAPTER 3
BACKGROUND ON IMITATION LEARNING

As in the previous section, we operate in the finite horizon MDP setting $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, P, r, \mu, T\}$. Imitation learning is a unique problem where the ground-truth reward or cost is unknown, but we are given expert demonstrations. The goal is to use expert demonstrations to learn a policy that performs as well as the expert, $\pi^\star$. Broadly, there are three settings for imitation learning which vary in level of access to the expert.

- **Offline:** $D^\star = \left\{(s^i, a^i)_{i=1}^N \sim d^{\pi^*}\right\}$. In this setting, we are only provided state-action pairs sampled from the expert and no interaction with the MDP is allowed.
- **Hybrid:** $D^\star = \left\{(s^i, a^i)_{i=1}^N \sim d^{\pi^*}\right\}$, known transitions. Beyond having the expert dataset, we also have access to the ground truth transition dynamics of the MDP.
- **Interactive:** $D^\star = \left\{(s^i, a^i)_{i=1}^N \sim d^{\pi^*}\right\}$ and known transitions, and $\pi^\star$ can now be queried. We are provided the same known transitions but now can query the expert policy at any state s during training to retrieve $\pi^\star(s)$.

We focus on the first two settings for this thesis. The following sections briefly describe the algorithms of interest. Note that in the original papers, the logarithm base used is not specified in most cases so unless otherwise specified, $\log = \ln$ in this thesis.

3.1 Behavior Cloning

Behavior cloning [21] is one of simplest offline imitation Learning algorithms. Assume we have a restricted policy class $\Pi = \{\pi : S \to \Delta(A)\}$ and that the expert $\pi^\star \in \Pi$. Behavior cloning learns a policy via a reduction to supervised learning:

$$\hat{\pi} = \operatorname*{argmin}_{\pi \in \Pi} \sum_{i=1}^{N} \ell(\pi, s^i, a^i) \tag{3.1}$$

where ℓ is a loss function. For example, if we use negative log-likelihood as the loss $\ell(\pi, s, a) = -\ln \pi(a|s)$, then our learning problem is reduced to the method of Maximum Likelihood Estimation (MLE) which can be solved using an optimization algorithm such as SGD or Adam [11].

Behavior cloning is a classical algorithm that has seen use dating all the way back to 1989 when it was used to develop ALVINN, shown in Figure 3.1.

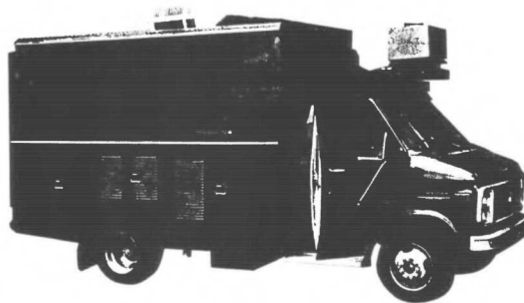


Figure 3.1: Image of ALVINN.

ALVINN is a vehicle that could be considered one of the forefathers of today's self-driving cars. Unfortunately, behavior cloning only ensures that the learned policy $\hat{\pi}$ does well under $d^{\pi^\star}$ so it suffers greatly from the issue of covariate shift. Even if supervised learning succeeds, we have with probability

$1 - \delta$ [37, 3] that:

$$\mathop{\mathbb{E}}_{s \sim d^{\pi^\star}} [\hat{\pi}(s) \neq \pi^\star(s)] \leqslant \frac{2 \ln(|\Pi|/\delta)}{N} \tag{3.2}$$

This tells us there will always be a small amount of error made by our learned policy. This leads to the following sample complexity bound:

$$V^\star - V^{\hat{\pi}} \leqslant 3T^2 \frac{2 \ln(|\Pi|/\delta)}{N} \tag{3.3}$$

Intuitively, this compounding error occurs because the states $\hat{\pi}$ makes predictions on in the environment come from $d^{\hat{\pi}}$, its own state-distribution, not $d^{\pi^\star}$

3.2 Generative Adversarial Imitation Learning

Another imitation learning algorithm that is relevant for our work is Generative Adversarial Imitation Learning (GAIL)[8] which operates in the hybrid setting. As the name implies, GAIL applies the concept of Generative Adversarial Networks (GAN) [6] to the imitation learning setting. To understand GAILs, a brief discussion about Maximum Entropy Inverse Reinforcement Learning (MaxEnt-IRL) [39] and GANs is required as GAIL builds upon these two concepts.

3.2.1 Maximum Entropy Inverse Reinforcement Learning

In MaxEnt-IRL, we are again in the hybrid setting and learn using costs rather than rewards. We assume that the true costs are linear in some feature mapping $\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^n$. Specifically, $c(s, a) = \langle \theta^*, \phi(s, a) \rangle$. Using the principle of maximum entropy, MaxEnt-IRL aims to choose a policy with the highest entropy

subject to a feature mapping constraint. Define $H(\pi) = \mathbb{E}_\pi[-\log(\pi(a|s))]$ as the causal entropy of the policy π. The MaxEnt-IRL optimization problem is:

$$\begin{aligned} &\underset{\pi}{\text{maximize}} \quad H(\pi) \\ &\text{subject to} \quad \mathop{\mathbb{E}}_{s,a\sim d^{\pi}} \phi(s,a) = \mathop{\mathbb{E}}_{s,a\sim d^{\pi^\star}} \phi(s,a) \end{aligned} \tag{3.4}$$

Using Lagrange multipliers, we can convert this into an optimization over the cost function parameters. We define the Lagrangian as:

$$L(\pi,\theta) = -H(\pi) - \left\langle \theta, \mathop{\mathbb{E}}_{s,a\sim d^{\pi^\star}} \phi(s,a) - \mathop{\mathbb{E}}_{s,a\sim d^{\pi}} \phi(s,a) \right\rangle$$

Noting that strong duality holds, we have the following equivalent optimization problem:

$$\max_{\theta} \min_{\pi} -H(\pi) + \mathop{\mathbb{E}}_{s,a\sim d^{\pi}} \langle \theta, \phi(s,a) \rangle - \mathop{\mathbb{E}}_{s,a\sim d^{\pi*}} \langle \theta, \phi(s,a) \rangle \tag{3.5}$$

The policy corresponding to the inner minimization can be computed by doing entropy regularized planning (e.g., soft value iteration/dynamic programming for the finite-horizon case). Taking the gradient with respect to θ gives us a gradient-based method to retrieve the cost function parameters. The update rule at iteration k is:

$$\theta_{k+1} = \theta_k + \eta[\mathop{\mathbb{E}}_{s,a\sim d^{\pi}_{\theta_k}} \phi(s,a) - \mathop{\mathbb{E}}_{s,a\sim d^{\pi*}} \phi(s,a)] \tag{3.6}$$

$\mathbb{E}_{s,a\sim d^{\pi}_{\theta_k}} \phi(s,a)$ can be estimated by sampling trajectories from the current policy while the latter can be estimated using $D^\star$ as $\frac{1}{|D^*|} \sum\limits_{s,a\in D^*} \phi(s,a)$.

3.2.2 Generative Adversarial Networks

The other concept that GAIL builds upon are GANs. GANs are a way to train a generative model by having two models, the generator and discriminator, compete against each other in an adversarial minimax two-player game. The generative model, G, is trained to capture the data distribution and the aim is to

generate examples that are indistinguishable from training samples by the discriminator, D.

Mathematically, let p_{data} and p_g represent the data and generator distribution, respectively. Before learning p_g, a prior on the input noise $p_z(z)$ is defined. $G(z;\theta_g)$, a multi-layer perception with parameters θ_g, is then defined as a mapping from the input noise to the data space. Similarly, $\mathcal{D}(x,\theta_d)$ maps data to a single scalar label that indicates whether the input data x is from the training data or generated from G. The objective is as follows:

$$\min_G \max_{\mathcal{D}} \mathop{\mathbb{E}}_{x\sim p_{\text{data}}}(x)[\log \mathcal{D}(x)] + \mathop{\mathbb{E}}_{z\sim p_z(z)}[\log(1-\mathcal{D}(G(z)))] \tag{3.7}$$

3.2.3 GAIL

Generative Adversarial Imitation Learning (GAIL) is an imitation learning algorithm that attempts to learn the policy directly rather than first learning a cost (or reward) before learning as in the case of inverse RL. To link GAIL to GAN, the policy can be considered as the generator and the discriminator attempts to distinguish between state-action pairs generated by the policy and those in the expert's visitation.

Let $C \in \mathbb{R}^{\mathcal{S}\times\mathcal{A}}$ be the set of all cost functions. To derive the GAIL objective, the IRL procedure is first defined as:

$$\underset{\psi}{\text{IRL}}(\pi^*) = \underset{c\in C}{\text{argmax}} - \psi(c) + \min_{\pi} -H(\pi) + \mathop{\mathbb{E}}_{s,a\sim d^{\pi}} c(s,a) - \mathop{\mathbb{E}}_{s,a\sim d^{\pi^*}} c(s,a)$$

The ψ term is a closed and convex cost function regularizer $C \to \mathbb{R} \cup \{\infty\}$. Given the large size of C, the IRL procedure can overfit to the provided data, so a fix is to regularize the cost function similar to the entropy term in MaxEnt-IRL. Given

the cost function, c, the RL procedure will do entropy regularized planning:

$$\mathrm{RL}(c) = \underset{\pi}{\operatorname{argmin}} - H(\pi) + \underset{s,a \sim d^{\pi}}{\mathbb{E}} c(s,a)$$

The resulting policy from the composition RL $\circ$ IRL actually corresponds to finding a policy whose visitation is close to the expert's visitation.

$$\mathrm{RL} \circ \underset{\psi}{\mathrm{IRL}}(\pi^*) = \underset{\pi}{\operatorname{argmin}} - H(\pi) + \psi^*(d^{\pi} - d^{\pi^*}),$$

ψ^* is the convex conjugate of the regularizer ψ defined as $\psi^*(c) = \sup_{c' \in \mathbb{R}^{\mathcal{S}\times\mathcal{A}}} c^T c' - \psi(c')$. The GAIL algorithm is derived by specifying the following choice of ψ:

$$g(x) = \begin{cases} -x - \log(1 - e^x) & x < 0 \\ \infty & \text{else} \end{cases} \qquad \psi_{GA}(c) = \begin{cases} \underset{s,a \sim d^{\pi^*}}{\mathbb{E}} g(c(s,a)) & c < 0 \\ \infty & \text{else} \end{cases}$$

Plugging this into $\mathrm{RL} \circ \mathrm{IRL}_{\psi}(\pi^*)$ gives us the following optimization problem for finding π:

$$\mathrm{RL}(\underset{\psi_{GA}}{\mathrm{IRL}}(\pi^*)) = \underset{\pi}{\operatorname{argmin}} - H(\pi) + \max_{\mathcal{D} \in (0,1)^{\mathcal{S}\times\mathcal{A}}} \underset{s,a \sim d^{\pi}}{\mathbb{E}} \log \mathcal{D}(s,a) + \underset{s,a \sim d^{\pi^*}}{\mathbb{E}} \log(1 - \mathcal{D}(s,a)) \tag{3.8}$$

The objective function essentially combines the objective functions for GANs and MaxEnt-IRL. In fact, this optimization problem is essentially the same as in MaxEnt-IRL except that the linear cost functions have been extended to deep neural networks. Taking inspiration from GANs, these neural networks can be considered discriminators that distinguish between samples from the expert dataset and samples generated by our policy.

JS-divergence minimization

An alternative way to look at this optimization problem is to look at it as minimizing the JS-divergence between the policy and expert's state-action distribu-

tion. To derive this alternative approach, we first make note that for a fixed π, the optimal discriminator is:

$$\frac{d^{\pi}(s,a)}{d^{\pi}(s,a)+d^{\pi^{\star}}(s,a)} = \operatorname*{argmax}_{\mathcal{D}\in(0,1)^{\mathcal{S}\times\mathcal{A}}} \mathop{\mathbb{E}}_{s,a\sim d^{\pi}} \log \mathcal{D}(s,a) + \mathop{\mathbb{E}}_{s,a\sim d^{\pi^*}} \log(1-\mathcal{D}(s,a))$$

The above is computed by noting that maximizing the total expression in this case can be achieved by maximizing each point-wise term inside the sum. Next, we define the JS-divergence and KL-divergence given two probability distributions P and Q defined on the probability space X:

$$\begin{aligned} KL(P,Q) &= \mathop{\mathbb{E}}_{x\sim X} \log(\frac{P(x)}{Q(x)}) \\ JSD(P,Q) &= \frac{1}{2}KL(P,\frac{P+Q}{2}) + \frac{1}{2}KL(Q,\frac{P+Q}{2}) \\ &= \frac{1}{2}KL(P,P+Q) + \frac{1}{2}KL(Q,P+Q) + \log(2) \end{aligned}$$

We can now rewrite the inner maximization problem of the GAIL objective as:

$$\begin{aligned} &\max_{\mathcal{D}\in(0,1)^{\mathcal{S}\times\mathcal{A}}} \mathop{\mathbb{E}}_{s,a\sim d^{\pi}} \log \mathcal{D}(s,a) + \mathop{\mathbb{E}}_{s,a\sim d^{\pi^*}} \log(1-\mathcal{D}(s,a)) \\ &= \mathop{\mathbb{E}}_{s,a\sim d^{\pi}} \log \frac{d^{\pi}(s,a)}{d^{\pi}(s,a)+d^{\pi^{\star}}(s,a)} + \mathop{\mathbb{E}}_{s,a\sim d^{\pi^*}} \log\left(1-\frac{d^{\pi}(s,a)}{d^{\pi}(s,a)+d^{\pi^{\star}}(s,a)}\right) \\ &= \mathop{\mathbb{E}}_{s,a\sim d^{\pi}} \log \frac{d^{\pi}(s,a)}{d^{\pi}(s,a)+d^{\pi^{\star}}(s,a)} + \mathop{\mathbb{E}}_{s,a\sim d^{\pi^*}} \log \frac{d^{\pi^{\star}}(s,a)}{d^{\pi}(s,a)+d^{\pi^{\star}}(s,a)} \\ &= KL\left(d^{\pi}, d^{\pi}+d^{\pi^{\star}}\right) + KL\left(d^{\pi^{\star}}, d^{\pi}+d^{\pi^{\star}}\right) \\ &= 2JSD\left(d^{\pi}, d^{\pi^{\star}}\right) - \log(4) \end{aligned}$$

The GAIL optimization problem can now be written as:

$$\begin{aligned} \mathrm{RL}(\underset{\psi_{GA}}{\mathrm{IRL}}(\pi^*)) &= \operatorname*{argmin}_{\pi} - H(\pi) + \max_{\mathcal{D}\in(0,1)^{\mathcal{S}\times\mathcal{A}}} \mathop{\mathbb{E}}_{s,a\sim d^{\pi}} \log \mathcal{D}(s,a) + \mathop{\mathbb{E}}_{s,a\sim d^{\pi^*}} \log(1-\mathcal{D}(s,a)) \\ &= \operatorname*{argmin}_{\pi} - H(\pi) + 2JSD\left(d^{\pi}, d^{\pi^{\star}}\right) - \log(4) \\ &= \operatorname*{argmin}_{\pi} - H(\pi) + 2JSD\left(d^{\pi}, d^{\pi^{\star}}\right) \qquad (3.9) \end{aligned}$$

This shows that the GAIL objective corresponds to finding a policy with the highest entropy that also minimizes the JS-divergence and expert's state-action

distribution. Besides the fact that the linear cost functions (which can be considered discriminators) have been extended to deep neural networks, this objective is identical to the MaxEnt-IRL, which finds a policy with the highest entropy subject to a feature mapping constraint. The algorithm for GAIL is shown below in Algorithm 3.1.

Algorithm 3.1 GAIL

1: **Input:** $D^* = \{\tau_E\}$ where $\tau_E^i = \{s_t^i, a_t^i, r_t^i\}_{t=0}^{T-1} \sim d^{\pi^*}$, policy parameters θ_0, discriminator parameters w_0, entropy regularization parameter λ

2: **for** $k = 0, 1, 2, \cdots$ **do**

3: Sample trajectories $D_k = \{\tau^i\}$ where $\tau^i = \{s_t^i, a_t^i, r_t^i\}_{t=0}^{T-1} \sim d^{\pi_{\theta_k}}$.

4: Update discriminator parameters from w_k to w_{k+1} with gradient $\mathbb{E}_{\tau^i}[\nabla_w \log(\mathcal{D}_{w_k}(s,a))] + \mathbb{E}_{\tau_E^i}[\nabla_w \log(1 - \mathcal{D}_{w_k}(s,a))]$. The following gradient estimate is used:

$$\frac{1}{T|D_k|}\sum_{i=1}^{|D_k|}\sum_{t=0}^{T-1}\nabla_w \log(\mathcal{D}_{w_k}(s_t^i, a_t^i)) + \frac{1}{T|D^\star|}\sum_{i=1}^{|D^\star|}\sum_{t=0}^{T-1}\nabla_w \log(1 - \mathcal{D}_{w_k}(s,a))$$

5: Update policy parameters θ_k to θ_{k+1} using TRPO rule with cost function $\log(\mathcal{D}_{w_{k+1}}(s,a))$. Take a KL-constrained natural gradient step

$$\mathbb{E}_{\tau^i}[\nabla_\theta \log \pi_{\theta_k}(a|s) Q(s,a)] - \lambda \nabla_\theta H(\pi_{\theta_k})$$

where $Q(s,a) = \mathbb{E}_{\tau^i}[\log(\mathcal{D}_{w_{k+1}}(s,a))|s_0 = s, a_0 = a]$ As usual, an estimate is used with the gradient step being:

$$\frac{1}{T|D_k|}\sum_{i=1}^{|D_k|}\sum_{t=0}^{T-1}\nabla_\theta \log \pi_{\theta_k}(a_t|s_t) Q(s_t, a_t) - \lambda \nabla_\theta H(\pi_{\theta_k})$$

6: **end for**

CHAPTER 4

BACKGROUND ON MODEL-BASED IMITATION LEARNING

Model-based Imitation Learning from Offline Data (MILO) is an algorithm developed by Jonathan Chang and other collaborators from Professor Sun's lab. This algorithm is the primary algorithm we tested in our research. In the following sections, we describe the motivation behind MILO, the algorithm itself, as well as some results from the original paper. The original authors also presented many theoretical results about MILO that will not be presented here as the thesis focuses on the practical application of MILO.

4.1 Motivation

As mentioned in the discussion about behavior cloning, covariate shift is a large issue in imitation learning. Prior methods mentioned in the paper to tackle covariate shift include:

- Interactive algorithms such as DAgger [26] or AggreVaTe [25, 32] can learn a policy that mimics the expert, but only when the expert is recoverable. That is, $\max_{s,a} A^{\pi^\star}(s, a) \leqslant c \in \mathbb{R}$ where c is some small constant.
- Online algorithms that require real-world interactions or a known dynamic model.
- Algorithms that assume the expert visits the entire state space.

MILO is an imitation learning algorithm that attempts to address the downsides of the above three methods. At a high level, MILO is a purely offline

imitation learning algorithm that doesn't require access to real-world dynamics. Additionally, access to the expert is not needed and the expert data does not need to cover the entire state space.

4.2 MILO Algorithm

The setting for MILO is the same setting that's been used previously. That is, we operate in a finite-horizon MDP, $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, P, c, \mu, T\}$. The cost function, c, is unknown as usual in an IL setting.

4.2.1 Datasets

As MILO is purely offline, careful consideration into the datasets is required. MILO uses two different datasets, D_e and D_o. $D_e = \{s_i, a_i\}_{i=1}^{n_e}$ is a dataset of n_e i.i.d samples collected from the expert, π^e, which does not need to cover the entire state space. $D_o = \{s_i, a_i, s_i^{'}\}_{i=1}^{n_o}$ is a dataset of n_o i.i.d samples collected from behavior policies which may be much worse than the expert, π^e. Specifically, (s, a) tuples are sampled from $\rho(s, a) \in \Delta(\mathcal{S} \times \mathcal{A})$ which is an offline distribution derived from behavior policies. The next state, s', is simply sampled from $P(s, a)$, the real world dynamics. The key thing to note about D_o is that only coverage of the expert's state-actions is required so $\max_{s,a} \frac{d^{\pi^e}(s,a)}{\rho(s,a)} < \infty$

The goal of MILO is to use these two datasets to learn a policy that performs as well as π_e. The hope is that this additional dataset, D_o, will help combat covariate shift. There are several key steps to MILO which are described in the following subsections.

4.2.2 Dynamics Model

The first step is to train a dynamics model, $\hat{P} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$, from D_o in order to learn the real-world dynamics P. In practice, an ensemble of neural networks is trained and is also used to construct the penalty function. The neural networks are simple MLPs that are trained using algorithms such as SGD or Adam.

4.2.3 Pessimistic Penalty Design

The penalty function, $b : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}^+$ is a function that gives high penalty to states and actions that are not covered by the offline data D_o and low penalty to those that the offline data does cover. The hope is that using this function in conjunction with traditional imitation learning will teach the policy to stay away from parts of the state-action space in which the dynamics model, $\hat{P}$, is highly inaccurate compared to the real-world. The dynamics model will be accurate for the state-action distribution of the offline dataset, which also should cover the expert's state-action distribution. In practice, $b(s,a)$ is equal to the max disagreement between all the models in the ensemble at that state and action. That is, $b(s,a) = \max_{i,j} \|\hat{P}_i(s,a) - \hat{P}_j(s,a)\|_2$.

4.2.4 Pessimistic Model-Based Min-Max Imitation Learning

The policy, π, is modeled as a neural network whose output is modeled as a Gaussian. This is typical for continuous control tasks. With the above steps

completed, the final step is to train the policy by solving:

$$\hat{\pi} = \operatorname*{argmin}_{\pi \in \Pi} \max_{f \in \mathcal{F}} \left[\mathop{\mathbb{E}}_{(s,a) \sim d^{\pi}_{\hat{P}}} [f(s,a) + b(s,a)] - \mathop{\mathbb{E}}_{(s,a) \sim D_e} [f(s,a)] \right] \tag{4.1}$$

where $d^{\pi}_{\hat{P}}$ is $\pi's$ state-action distribution in the dynamics model $\hat{P}$ and $\mathcal{F}$ is the cost-function class. The functions in $\mathcal{F}$ can also be thought of as a discriminator as the goal of f is to disambiguate state-action pairs sampled from $d^{\pi}_{\hat{P}}$ and those from D_e. The maximization inside the objective corresponds to using the Integral Probability Metric (IPM) as a distance metric. To be exact, given two probability measures, $\mathcal{X}$ and $\mathcal{Y}$, which are both defined on a measurable space, $\mathcal{S}$, and $\mathcal{F}$, a class of real-valued bounded measurable functions on $\mathcal{S}$, the IPM is defined as:

$$d_{\mathcal{F}}(\mathcal{X}, \mathcal{Y}) = \sup_{f \in \mathcal{F}} \left| \int_{S} f d\mathcal{X} - \int_{S} f d\mathcal{Y} \right|$$

In our specific case, given two state-action distributions d_1 and d_2, the IPM is:

$$d_{\mathcal{F}}(d_1, d_2) = \max_{f \in \mathcal{F}} \left[\mathop{\mathbb{E}}_{(s,a) \sim d_1} [f(s,a)] - \mathop{\mathbb{E}}_{(s,a) \sim d_2} [f(s,a)] \right]$$

There are various distance metrics that are derived from the above equation by specifying $\mathcal{F}$. Examples include the Dudley Metric, Wasserstein distance, and Maximum Mean Discrepancy [7].

Adding regularization to the IL objective

In order to ensure incremental policy updates (like in NPG or TRPO), a KL-based trust-region formulation for incremental policy update inside the dynamics is used by adding a constraint to the objective:

$$\begin{aligned} &\operatorname*{argmin}_{\pi \in \Pi} \max_{f \in \mathcal{F}} \left[\mathop{\mathbb{E}}_{(s,a) \sim d^{\pi}_{\hat{P}}} [f(s,a) + b(s,a)] - \mathop{\mathbb{E}}_{(s,a) \sim D_e} [f(s,a)] \right] \\ &\text{subject to} \quad \mathop{\mathbb{E}}_{(s,a) \sim D_e} [\ell(a, s, \pi)] \leqslant \delta \end{aligned} \tag{4.2}$$

$\ell : \mathcal{A} \times \mathcal{S} \times \Pi \rightarrow \mathbb{R}$ in the constraint is a loss function (like NLL used in behavior cloning). In practice, the hard constraint is replaced by a Lagrange multiplier.

$$\underset{\pi \in \Pi}{\operatorname{argmin}} \max_{f \in \mathcal{F}} \left[\mathop{\mathbb{E}}_{(s,a) \sim d^{\pi}_{\hat{P}}} [f(s,a) + b(s,a)] - \mathop{\mathbb{E}}_{(s,a) \sim D_e} [f(s,a)] \right] + \lambda \mathop{\mathbb{E}}_{(s,a) \sim D_e} [\ell(a,s,\pi)] \quad (4.3)$$

This behavior cloning term essentially serves as regularization.

4.2.5 Cost Function

For the distance metric, the Maximum Mean Discrepancy (MMD) is used. In this case, $\mathcal{F} = \{f : \|f\|_{\mathcal{H}} \leqslant 1\}$ where $\mathcal{H}$ is a reproducing kernel Hilbert space (RKHS). In MILO, the kernel of interest is the Radial Basis Function kernel. As is often the case when using kernels with large datasets, Random Fourier Features (RFF)[22] are used instead to approximate the kernel. RFF is a widely used technique for scaling up kernel methods through approximation, as using exact kernels does not scale well. Let $k((s,a),(s',a'))$ represent the RBF kernel and $\phi(s,a)$ represent RFF:

$$k((s,a),(s'a')) \approx \phi^{\top}(s,a)\phi(s',a')$$

The cost function class, $\mathcal{F}$, becomes the set of linear classifiers defined by $f(s,a) = w^{\top}\phi(s,a)$ where w is the vector of parameters. The closed form solution of the inner maximization problem can be computed exactly due to using the MMD cost. Plugging in $f(s,a) = w^{\top}\phi(s,a)$ into the objective and placing constraint on the size of w gives:

$$\max_{w : \|w\|_2^2 \leqslant \eta} \left[\mathop{\mathbb{E}}_{(s,a) \sim d^{\pi}_{\hat{P}}} [w^{\top}\phi(s,a) + b(s,a)] - \mathop{\mathbb{E}}_{(s,a) \sim D_e} [w^{\top}\phi(s,a)] \right] - \frac{1}{2}(\|w\|_2^2 - \eta) \quad (4.4)$$

The closed form solution at iteration k is:

$$w_k = \mathop{\mathbb{E}}_{(s,a) \sim d^{\pi}_{\hat{P}}} [\phi(s,a)] - \mathop{\mathbb{E}}_{(s,a) \sim D_e} [\phi(s,a)]$$

The closed form solution matches intuition as MMD measures the distance between mean embeddings in $\mathcal{H}$. $\mathbb{E}_{(s,a)\sim d^{\pi}_{\hat{P}}}[\phi(s,a)]$ is estimated by collecting samples from $d^{\pi}_{\hat{P}}$ by rolling out the current policy, π_{θ_k} in the environment $\hat{P}$. The cost is updated to be:

$$c_k(s,a) = (1-\lambda_{\text{penalty}})f_k(s,a) + \lambda_{\text{penalty}}b(s,a) \tag{4.5}$$

$$= (1-\lambda_{\text{penalty}})w_k^{\top}\phi(s,a) + \lambda_{\text{penalty}}b(s,a) \tag{4.6}$$

where λ_{penalty} is used to balance the penalty with the cost term. With this cost function, traditional policy gradient methods can be used to find π_k.

4.2.6 Updating the Policy

As the policy is represented using neural networks, it is natural to use policy gradient algorithms such as those in chapter 3. The algorithm used in MILO is NPG or TRPO.

At iteration k with polity π_{θ_k}, the first step is compute the dis-advantage function, $A^{\pi_{\theta_t}}_{\hat{P},c_t}$, using the trajectories collected from rolling out π_{θ_t} in $\hat{P}$. $A^{\pi_{\theta_t}}_{\hat{P},c_t}$ is called the dis-advantage function since MILO deals with cost, not reward so a positive value means worse performance when taking the action a. Then, the Fisher information matrix is computed:

$$F_{\theta_t} = \mathbb{E}_{(s,a)\sim d^{\pi_{\theta_t}}_{\hat{P}}}[\nabla \ln \pi_{\theta_t}(a|s) \nabla \ln \pi_{\theta_t}(a|s)^{\top}]$$

The policy is updated using an NPG or TRPO update. Note that the optimization problem is minimization since we're dealing with cost. The update rule at iteration k is:

$$\theta_{k+1} = \theta_k - \eta F_{\theta_k}^{-1}\left(\mathbb{E}_{(s,a)\sim d_{\hat{P}^{\pi_{\theta_k}}}}\left[\nabla \ln \pi_{\theta_k}(a|s) A^{\pi_{\theta_k}}_{\hat{P},c_t}(s,a)\right] + \lambda \mathbb{E}_{(s,a)\sim D_e}[\nabla \ell(a,s,\pi_{\theta_k})]\right) \tag{4.7}$$

A high-level overview of the algorithm is shown in Algorithm 4.1.

Algorithm 4.1 MILO

1: **Input:** policy parameters θ_0, value function parameters ψ_0, NPG parameters, expert dataset $D_e = \{s_i, a_i\}_i^{n_e}$, offline dataset $D_o := \{s_i, a_i, s_i'\}_i^{n_o}$
2: Train an ensemble of neural networks $\{\hat{P}_1, \cdots, \hat{P}_n\}$ where each model starts with random initialization using D_o.
3: Set the bonus function $b(s, a) = \max_{i,j} \|\hat{P}_i(s, a) - \hat{P}_j(s, a)\|_2$.
4: **for** $k = 0, 1, 2, \cdots$ **do**
5: Set the current dynamics model to be $\hat{P} = \hat{P}_{(k \bmod n)}$.
6: Collect set of trajectories $D_k = \{\tau^i\}$ where $\tau^i = \{s_h^i, a_h^i\}_{h=0}^{H-1}$ by rolling out using $\pi_k = \pi(\theta_k)$ in $\hat{P}$.
7: Solve for the optimal discriminator parameters, w_k by solving:

$$w_k = \underset{\|w\|_2 \leqslant 1}{\operatorname{argmax}} \, w^\top \left(\mathop{\mathbb{E}}_{(s,a) \sim d_{\hat{P}}^{\pi_{\theta_k}}} [\phi(s, a)] - \mathop{\mathbb{E}}_{(s,a) \sim D_e} [\phi(s, a)] \right)$$

8: Form the optimal discriminator $f_k(s, a) = w^\top \phi(s, a)$ and cost function $c_k(s, a) = f_k(s, a) + b(s, a)$.
9: Compute the costs for the trajectories in D_k using c_k and compute dis-advantage esti-. mates, $A_{\hat{P}, c_k}^{\pi_{\theta_k}}$, using any advantage estimate algorithm such as GAE based on V_{ψ_k}
10: Compute the step size, η, for an NPG or TRPO step and update θ_t as:

$$\theta_{k+1} = \theta_k - \eta F_{\theta_k}^{-1} \left(\mathop{\mathbb{E}}_{(s,a) \sim d_{\hat{P}}^{\pi_{\theta_k}}} \left[\nabla \ln \pi_{\theta_k}(a|s) A_{\hat{P}, c_k}^{\pi_{\theta_k}}(s, a) \right] + \lambda \mathop{\mathbb{E}}_{(s,a) \sim D_e} [\nabla \ell(a, s, \pi_{\theta_k})] \right)$$

11: Fit value-function using regression on MSE loss

$$\psi_{k+1} = \underset{\psi}{\operatorname{argmin}} \frac{1}{2|D_k|T} \sum_{i=0}^{|D_k|-1} \sum_{t=0}^{T-1} (y^i - V_{\psi_k}(s_t^i))^2$$

12: **end for**

4.3 Experiments

This section briefly describes the experimental results shown in the original paper, which will be useful as a point of comparison when analyzing our results in later chapters. The experiments in the MILO paper were done using several OpenAI Gym environments using MuJoCo [35]. For the RL algorithm implementations, MILO makes use of the open-source MJRL [24] package, which contains implementations of various RL algorithms for continuous control tasks

in MuJoCo.

The expert dataset for these tasks consisted of random (s-a)-pairs sampled from 100 expert trajectories collected from expert policies trained using traditional RL methods. The random sampling was done to ensure BC had a tough time learning. The offline dataset consisted of samples from behavior policies whose performance were worse than the expert's. Figure 4.1 shows how MILO performed against other algorithms.

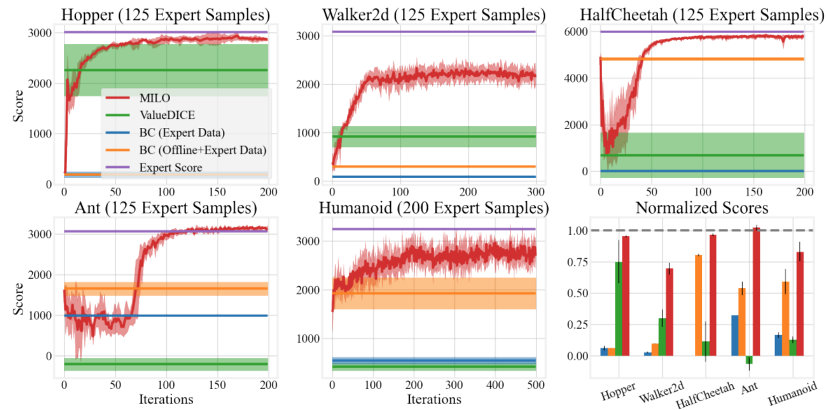


Figure 4.1: This figure is taken directly from the MILO paper experiments section (figure 2). It shows learning curves across five seeds for MILO against BC after 1000 epochs of training and ValueDICE after 10 thousand iterations. The normalized graph shows the normalized scores for the expert.

Other results showed that adding the penalty cost function did indeed help learning. Additionally, adding more expert samples into the expert dataset also improved learning. Finally, it was shown that performance decreases the less coverage there is of the expert's state-action space in the offline dataset.

CHAPTER 5

BACKGROUND ON DEEPMIMIC AND AMP

DeepMimic and its extension, AMP, are part of a framework developed for utilizing reinforcement learning in order to train control policies for simulated characters. The framework consists of custom characters and uses reference motion clips for training. The goal of the framework is to imitate reference motions while also accomplishing goals such as striking or throwing. The following sections discuss the background for the framework as well as the extensions AMP makes upon the framework.

5.1 DeepMimic

DeepMimic is a framework, written in C++ and Python, for teaching simulated characters how to imitate reference motions while satisfying certain goal objectives. At a high-level, the system receives as input a character model, a set of reference motions, and a goal-defined reward function and outputs a controller for the character. Since our work focuses mainly on imitation objective, the goals will only be briefly discussed.

5.1.1 The Characters

The characters that were tested in the original paper were a 3D humanoid, an Atlas robot model, a T-Rex, an a dragon as shown in Figure 5.1.

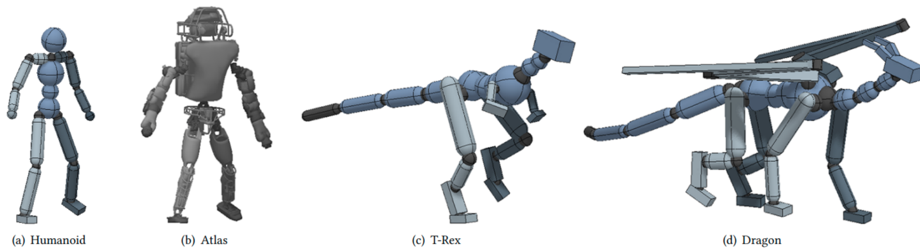


Figure 5.1: Figures of the four characters in the DeepMimic and AMP framework.

These models are rigid bodies with 3 degree-of-freedom (DOF) spherical joints connecting most links except the knees and elbows where 1 DOF revolute joints are used. PD controllers are placed at the joints to control each character. All these characters vastly different in either body shape, mass, PD gains and torque limits. Additionally, the characters have much larger space and action spaces compared to characters in frameworks such as MuJoCo. The authors were able to show that their method could be applied to a wide range of characters.

The reference motions, $\{\hat{q}_t\}$, are motion-capture clips for the humanoid and Atlas robot, while custom artist-created keyframes are used for the T-Rex and dragon. Regardless of the character, the reference motion files follow a JSON format in which a ”Loop” field specifies whether the motion is cyclic while the ”Frames” field is a list of vectors for the keyframes. Each vector contains information about the duration of the frame, the world position and world rotation of the root (in meters), 3D local rotations for the spherical joints as quaternions, and revolute joints as scalar rotations in radians. The frequency of these keyframes is 60Hz.

5.1.2 Simulating the World

In order to simulate the world, DeepMimic uses the Bullet Physics [1] library. This is a real-time collision detection and multi-physics simulator. The physics simulation is performed at 1.2kHz.

5.1.3 Framing the RL problem

Creating a controller that allows a character to imitate motion while satisfying goal objectives is framed as a reinforcement learning problem. Many of the previously mentioned algorithms will be used here. As typical, the MDP is defined as $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, P, r, \mu, T\}$

States, Actions, and Goals

The states represent the configuration of the character's body. Specifically, the state contains the world position of the root joint, the relative position of each link (including the root link) with respect to the root, their rotations, and their linear and angular velocities. The features are computed in the character's local frame. While the rotations are stored as quaternions in the code, the rotations are represented in the state by the norm and tangent vectors. Specifically, the norm and tangent vectors are the vectors $(0, 1, 0)$ and $(1, 0, 0)$, respectively, rotated by the quaternion. Finally, the state contains a 1D phase variable $\phi \in$ [0,1] that represents how far along the reference motion the character should be.

The character can be trained to fulfill specific goals along with the imitation of the reference motion. These goals include heading towards a target direction,

striking a target, throwing an object, and terrain traversal on uneven terrain. Depending on the goal, extra information is concatenated onto the state in the form of a goal vector. This goal vector can contain information such as the target direction, the location of the target, etc. If imitation is the only objective with no specific goal to achieve, then the goal vector is empty. In the case of attempting to imitate multiple motions with no goal, the goal is simply a one-hot vector representing which motion to imitate.

Each action specifies a target orientation for the PD controllers located at each joint. The spherical joint target orientations are represented by transforming the quaternion angle to 4D axis-angle form, while the revolute joint targets are represented by scalar rotation angles.

As is typical for many continuous control problems, the policy is represented as a neural network whose output is modeled as a Gaussian. However, the input is now dependent on both the state and the goal vector.

Reward Function

The rewards in DeepMimic are state-dependent but not action-dependent. The reward function at time t, r_t, has the following form:

$$r_t = \omega^I r_t^I + \omega^G r_t^G \tag{5.1}$$

where r_t^I is the imitation objective reward and r_t^G is the goal objective. ω^I and ω^G are their weights. The imitation reward can be broken down into further sub-rewards that give rewards for matching joint orientations and velocities of

the reference motions:

$$
\begin{aligned}
r_t^I &= w^P r_t^P + w^v r_t^v + w^e r_t^e + w^c r_t^c \\
r_t^p &= \exp\left[-2\left(\sum_j \|\hat{q}_t^j \ominus q_t^j\|^2\right)\right] \\
r_t^v &= \exp\left[-0.1\left(\sum_j \|\hat{\dot{q}}_t^j - \dot{q}_t^j\|^2\right)\right] \\
r_t^e &= \exp\left[-40\left(\sum_e \|\hat{p}_t^e - p_t^e\|^2\right)\right] \\
r_t^c &= \exp[-10(\|\hat{p}_t^c - p_t^c\|^2)]
\end{aligned}
$$

where $\ominus$ represents quaternion difference and $\|q\|$ is the scalar rotation of quaternion q about its axis in radians. The pose reward r_t^p is based on the pose similarity between the simulated character and reference motion. Specifically, it computes the quaternion difference between the orientations of the joints of the simulated character and reference motion. The velocity reward r_t^v is based on the difference in joint angular velocities of the simulated and reference motion. The end-effector reward r_t^e is based on how close the feed and hands of the character are to the reference motion. Each position is represented in 3D world position. Finally, the center-of-mass reward r_t^c is based on how close the character's center of mass is to the reference.

The other component of the reward r_t is the goal objective reward r_t^G. These rewards are hand-designed for the goal (e.g., target heading, throwing, etc.). As the work done in this thesis focuses on the imitation side, there is no goal so $r_t^G = 0$

5.1.4 The Learning Algorithm

Since the policy is parameterized by parameters θ, DeepMimic makes use of policy gradient algorithms. DeepMimic makes use of a goal-conditioned reinforcement learning objective. Given a goal $g \sim p(g)$ and a trajectory $\tau \sim \rho(\tau|\pi, g)$, the objective is to maximize:

$$\max_{\theta} \mathop{\mathbb{E}}_{p(g)} \mathop{\mathbb{E}}_{\rho(\tau|\pi_\theta, g)} \left[\sum_{t=0}^{T-1} \gamma^t r_t \right] \tag{5.2}$$

PPO is used to train policies. The value function, $V_\psi(s, g)$, is also parameterized as a neural network and TD(λ) is used to update the value function. GAE(γ, λ) is used along with $V_\psi(s, g)$ to get advantage estimates. Note that if there is no goal (i.e., only imitation), then the objective above is the same as in typical policy gradient algorithms.

Reference State Initialization and Early Termination

When sampling the start state, DeepMimic makes use of random state initialization (RSI). Rather than starting at the same fixed initial state, a character is reset to a random point in time in the reference motion. Typically, this is difficult to do in many situations, such as when working with physical robots. However, with simulated characters, it is trivial to reset to a random point in time. This has the advantage of aiding exploration. In addition, rather than forcing the policy to learn the motion sequentially, the policy can make progress in different phases of the motion. This is especially helpful when imitating complex motions such as spinkicks or backflips where sequential learning would discourage the character from jumping since it wouldn't know how to land, leading to worse returns.

While cyclic motions technically last for an infinite time, a finite horizon is set during training. Additionally, DeepMimic also ends an episode early whenever certain links of the character, such as the torso or hands, collide with the ground. This collision detection is handled by Bullet Physics. Early termination can greatly help training as without it, samples in early stages of training will consist mostly of character flailing on the ground which impedes learning. The training procedure for DeepMimic is summarized below. It follows mostly the same format as the PPO algorithm shown in Algorithm 2.2.

Algorithm 5.1 DeepMimic

1: **Input:** initial policy parameters θ, initial value function parameters ψ
2: **while** not done **do**
3: Sample initial state s_0 from reference motion using RSI and initialize simulated character to s_0.
4: Roll out episodes with π_θ in the Bullet Physics environment until m samples are collected. Each episode is rolled out until the horizon or early termination. Store all the (s, a, r, s') tuples in memory D.
5: $\theta_{old} \leftarrow \theta$
6: **for** each update step **do**
7: Sample mini-batch of samples $\{s_i, a_i, r_i, s'_i\}_{i=1}^{n}$ from D.
8: For each sample (s_i, a_i, r_i, s'_i), compute target y_i for updating V_ψ using TD(λ).
9: Update the value function with MSE loss.

$$\psi \leftarrow \psi + \alpha_v(\frac{1}{n}\sum_i \nabla_\psi V_\psi(s_i)(y_i - V_\psi(s_i)))$$

10: For each sample, (s_i, a_i, r_i, s'_i), compute advantage estimates, $\hat{A}_i$, using V_ψ and GAE(γ, λ).
11: **end for**
12: Update θ with mini-batch SGD using the following update rule:

$$\theta \leftarrow \theta + \alpha_\pi \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta \min\left(\frac{\pi_\theta(a_i|s_i)}{\pi_{\theta_{old}}(a_i|s_i)}\hat{A}_i, clip(\frac{\pi_\theta(a_i|s_i)}{\pi_{\theta_{old}}(a_i|s_i)}, 1-\epsilon, 1+\epsilon)\hat{A}_i\right)$$

13: **end while**

5.1.5 Goals and other extensions

Since the thesis work (to be described in chapter 8) focuses on imitation learning, we do not use the goals. However, given that much of the work of Deep-

Mimic (and AMP) involves these goals, this section will briefly describe the goals that policies can be trained to perform alongside motion imitation. The goals are described below:

- Striking: The goal here is for the character to strike a randomly placed spherical target using specific links. The goal vector that is appended onto the state contains target location and a binary variable indicating whether the target has been hit in a previous timestep.
- Throwing: The goal is to throw a ball to the target rather than striking. The goal vector is the same as in the striking goal, but the state contains the position, rotation, and the linear and velocity of the ball.
- Target Heading: The goal is to perform the motion in a certain direction so the target direction is appended onto the state.
- Terrain Traversal: Rather than performing the motion in a flat world, the character is trained to traverse different environments with obstacles or uneven terrain. For this goal, a visuomotor policy network is used. The policies take in as additional input a heightmap and use convolutional layers to aid with the vision goal. The goal vector is the same as the target-heading goal except the target heading is fixed forward so the character moves forward through the terrain.

DeepMimic also implements several extensions that utilize several reference motions at once, such as training a policy to imitate several different reference motions without any additional goal. In this case, the goal vector becomes a one-hot vector representing the motion to be imitated rather than additional information regarding the goal. Another extension is augmenting the policy to

allow the user to control which specific motion the character executes. This is assuming the policy is trained on various motion clips. Thus, the policy needs to not only be able to decide what motion to imitate in any given situation, but it also needs to be able to execute arbitrary skills specified by the users. However, the more motion clips, and subsequently skills, the policy needs to learn, the less likely it is that a single policy can learn all of them well. The use of a phase variable makes learning from multiple motion clips difficult. Thus, an alternative is to use a composite policy in which multiple policies, each trained on different skills, are combined. However, this can still be clunky and difficult to train. Having one policy be able to effectively learn from multiple motion clips without having to worry about explicit synchronization served as inspiration for the follow-up work, AMP.

5.2 AMP

Adversarial Motion Priors (AMP), is an extension upon the DeepMimic framework and directly adds to the existing DeepMimic codebase. AMP utilizes a fully automated approach for motion selection through adversarial imitation learning, removing the need for a manually designed imitation objective. In DeepMimic, a phase variable is needed in order to synchronize the character with a target pose which makes it especially suited for single-motion imitation. However, the use of a phase variable can make it difficult to scale to datasets that have various motions since a phase variable is often not enough to have proper synchronization. AMP uses adversarial imitation learning to teach characters to mimic the behaviors in the dataset, so it is much more suited for large datasets. Additionally, there is no hard constraint enforcing the character to ex-

actly match any one motion, allowing the character to instead mimic motions that match the ”style” defined by the dataset. For the remainder of this section, $\mathcal{N}$ will refer to the dataset of reference motions.

AMP and DeepMimic largely follow the same procedure, but AMP adds one additional step in the training process which is the adversarial imitation learning. In the following sections, we discuss the extensions AMP adds upon DeepMimic.

5.2.1 States, Actions, and Goals

The state in AMP is identical to the state in DeepMimic except that the phase variable has been removed. Thus, there is no explicit synchronization variable as this will be handled by adversarial imitation learning. The action is also similar to the representation in DeepMimic except that the target orientations for the spherical joints now are stored using 3D exponential maps instead of the 4D axis-angle form like in DeepMimic. Specifically, given a 3D exponential map q, the rotation axis v and angle θ can be computed as:

$$v = \frac{q}{\|q\|_2}$$

$$\theta = \|q\|_2$$

The goals are the same as in DeepMimic.

5.2.2 Reward Function

Like in DeepMimic, the reward function in AMP is defined using two components:

$$r(s_t, a_t, s_{t+1}, g) = \omega^G r^G(s_t, a_t, s_{t+1}, g) + \omega^S r^S(s_t, s_{t+1}) \tag{5.3}$$

r^G is the goal-specific reward (e.g., throwing, striking, etc.) as in DeepMimic. r^S can be considered as the "style" reward as it determines how the character performs the reward (e.g., running vs walking). Unlike the imitation objective reward r^I in DeepMimic, r^S is a learned goal-agnostic reward. Adversarial imitation learning is used to learn a r^S that allows the character to have natural behaviors.

Specifying the Discriminator

The training method for r^S uses a GAIL-style discriminator. As mentioned in section 3.2.3, GAIL utilizes states and actions but the issues is that the expert dataset in AMP consists of reference motions so there are no actions, only states, also called demonstrations or observations. Thus, AMP makes use of a version of GAIL that utilizes only observations and no actions called Generative Adversarial Imitation from Observation (GAIfO) [36]. The discriminator objective for GAIfO is:

$$\underset{\mathcal{D}}{\operatorname{argmax}} \ \mathbb{E}_{d^{\mathcal{M}}(s,s')}[\log(\mathcal{D}(s, s'))] + \mathbb{E}_{d^{\pi}(s,s')}[\log(1 - \mathcal{D}(s, s'))]$$

Instead of using the typical sigmoid cross-entropy loss used in GAN and GAIL which can lead to vanishing gradients and other optimization challenges,

AMP uses least-squares loss for GANs which was first introduced in least-squares GAN (LSGAN)[16]. In the paper for LSGANs, this loss has been shown to improve training and lead to better images for the traditional GAN image-synthesis task. As we showed earlier, when the discriminator is optimal in GAN, the loss function measures the JS divergence between the generative and expert dataset. Similarly, LSGAN's loss function measures the Pearson χ^2 divergence. Combining the ideas from GAIfO and LSGANs, the discriminator objective for AMP is:

$$\underset{\mathcal{D}}{\operatorname{argmin}} \underset{d^{\mathcal{N}}(s,s')}{\mathbb{E}}\left[(\mathcal{D}(s,s')-1)^2\right] + \underset{d^{\pi}(s,s')}{\mathbb{E}}\left[(\mathcal{D}(s,s')+1)^2\right]$$

where $\mathcal{D}(s,s') = 1$ indicates that the discriminator fully believes that the input states are from the reference motion while $\mathcal{D}(s,s') = -1$ indicates the discriminator thinks the states are generated from the policy. Note that this objective is a minimization.

Discriminator features

Rather than directly using the states for the discriminator, AMP uses an observation map $\Phi(s)$ to extract features for the discriminator. While the original state dealt with the links of the characters, the features for the discrinators describe the joints of the character. These include the linear and angular velocity of the root, the local rotation and velocity of each joint, and the 3D positions of the end-effectors (e.g., hands and feet). These features are extracted by querying the AMP framework, so they aren't computed in an offline matter. However, it should be possible to extract these features from the state vector solely and this is to be explored in future work.

Final Discriminator Objective

The final discriminator objective has the form:

$$\underset{\mathcal{D}}{\operatorname{argmin}} \underset{d^{\mathcal{N}}(s,s')}{\mathbb{E}}\left[(\mathcal{D}(\Phi(s),\Phi(s'))-1)^2\right] + \underset{d^{\pi}(s,s')}{\mathbb{E}}\left[(\mathcal{D}(\Phi(s),\Phi(s'))+1)^2\right] + \frac{w^{gp}}{2} \underset{d^{\mathcal{N}}(s,s')}{\mathbb{E}}\left[\|\nabla_{\phi}\mathcal{D}(\phi)|_{\phi=(\Phi(s),\Phi(s'))}\|^2\right] \quad (5.4)$$

where the third term $\frac{w^{gp}}{2} \underset{d^{\mathcal{N}}(s,s')}{\mathbb{E}}\left[\|\nabla_{\phi}\mathcal{D}(\phi)|_{\phi=(\Phi(s),\Phi(s'))}\|^2\right]$ is a gradient penalty used for stabilizing training [17] as naive GAN implementations can result in oscillatory behavior. Assuming the generator and discriminator are powerful enough (i.e., have large enough capacity), the goal of the GAN (or GAIL) algorithm is to find the Nash-equilibrium of the objective. In the traditional GAN objective, the Nash-equilibrium occurs when the generator matches the true data distribution exactly while the discriminator becomes the zero discriminator which outputs 0 everywhere on the data distribution.

Unfortunately, naive gradient-based algorithms used for GAN optimization often don't converge to this equilibrium. [17] argue that when the generator is near the true data distribution or even equal to it, there is nothing stopping the discriminator from having non-zero gradients orthogonal to the tangent space of the true data manifold. These non-zero gradients can push the generator away from the true distribution making it so that the generator doesn't converge. However, when the generator moves away from the true data distribution, the discriminator will be more certain on predicting the difference between the generated data and true data, thus encouraging the generator to move back to the true data distribution. This cycle can repeat endlessly, leading to very unstable training near the equilibrium point. Similarly, if the discriminator is close to the equilibrium discriminator, there is nothing to push the generator back to-

wards the true distribution since the equilibrium discriminator always outputs 0. Thus, when either the generator or discriminator are near equilibrium point, they don't produce useful gradients, meaning convergence is difficult.

According to [17], the issue with the discriminator producing non-zero gradients orthogonal to the tangent space of the true data manifold is especially prevalent when the data distribution is a low-dimensional manifold. If the class of discriminators is large enough, then these non-zero gradients are more likely to occur. A way to get around this is to add the gradient penalty shown in the earlier objective. This extra penalty penalizes gradients only on the real data. Thus, if we're at the Nash equilibrium where the generator produces the true data distribution and the discriminator predicts 0 for this data, the penalty makes it so that producing a non-zero gradient at these data points means the discriminator suffers in the objective.

The Style Reward

Once the discriminator is trained, the style reward r_t^G can be computed as:

$$r(s_t s_{t+1}) = \max[0, 1 - 0.25(\mathcal{D}(\Phi(s_t), \Phi(s_{t+1})) - 1)^2] \tag{5.5}$$

5.2.3 The Learning Algorithm

The pseudocode for AMP is shown in Algorithm 5.2.

Algorithm 5.2 AMP

1: **Input:** $\mathcal{N}$ dataset of reference motions, initial policy parameters θ, initial value function parameters ψ, initial discriminator parameters ω
2: Initialize replay buffer $\mathcal{B} \leftarrow \emptyset$
3: **while** not done **do**
4: **for** trajectory $i = 1, \cdots, m$ **do**
5: Collect trajectory $\tau^i = \{(s_t, a_t, r_t^G)_{t=0}^{T-1}, s_T^G, g\}$ by running π in the environment.
6: **for** timestep $t = 0, \cdots, T-1$ **do**
7: Compute $r_t^S(s_t, s_{t+1}) = \max[0, 1 - 0.25(\mathcal{D}(\Phi(s_t), \Phi(s_{t+1})) - 1)^2]$.
8: Compute $r_t = w^G r_t^G + w^S r_t^S$ and store r_t in τ_i.
9: **end for**
10: store τ^i in replay buffer $\mathcal{B}$
11: **end for**
12: **for** update step = $1, \cdots, n$ **do**
13: $b^{\mathcal{N}} \leftarrow$ sample batch of K transitions $\{(s_j, s'_j)\}_{j=0}^{K-1}$ from N
14: $b^{\pi} \leftarrow$ sample batch of K transitions $\{s_j, s'_j)\}_{j=0}^{K-1}$ from $\mathcal{B}$
15: Update the discriminator using the update rule:

$$\omega \leftarrow \underset{\omega}{\text{argmin}} \underset{d^{\mathcal{N}}(s,s')}{\mathbb{E}}[(\mathcal{D}_\omega(\Phi(s), \Phi(s')) - 1)^2] + \underset{d^{\pi}(s,s')}{\mathbb{E}}[(\mathcal{D}_\omega(\Phi(s), \Phi(s')) + 1)^2] + \frac{w^{gp}}{2} \underset{d^{\mathcal{N}}(s,s')}{\mathbb{E}}\left[\|\nabla_\phi \mathcal{D}_\omega(\phi)|_{\phi=(\Phi(s),\Phi(s'))}\|^2\right]$$

16: **end for**
17: Update the value-function and π using samples from $\{\tau^i\}_{i=1}^m$ following the procedure in DeepMimic.
18: **end while**

CHAPTER 6

DEVELOPING THE FRAMEWORK AND EXPERIMENTATION

As mentioned in the abstract and introduction, one of the primary goals of this project was to create an easy-to-use interface for interacting with the AMP framework. Specifically, we aimed to adapt the code AMP framework to fit into the OpenAI Gym framework. Once we did that, we could test MILO on AMP as much of the code used in MILO was based on MJRL, which used OpenAI Gym environments. Even though MJRL was originally meant for MuJoCo, we could easily adapt it to work with the AMP framework. For the remainder of this chapter, mention of the AMP framework refers to the codebase for both DeepMimic and AMP.

6.1 Setting up AMP

The AMP framework was first setup by following the instructions found `https://github.com/xbpeng/DeepMimic`. Due to the heavy use of C++ and TensorFlow (v1)[2], a large portion of time was devoted to setting up and understanding the codebase.

6.1.1 DeepMimicCore

All the C++ code is contained inside the directory `DeepMimicCore`. As the name of this folder suggests, `DeepMimicCore` contains all the essential code for the simulation of the world, characters, etc. Specifically, it handles creation of

the world and characters in Bullet Physics [1], the simulation steps, controlling the characters, drawing and rendering, and contains classes for imitation and different goals.

6.1.2 SWIG Wrapper

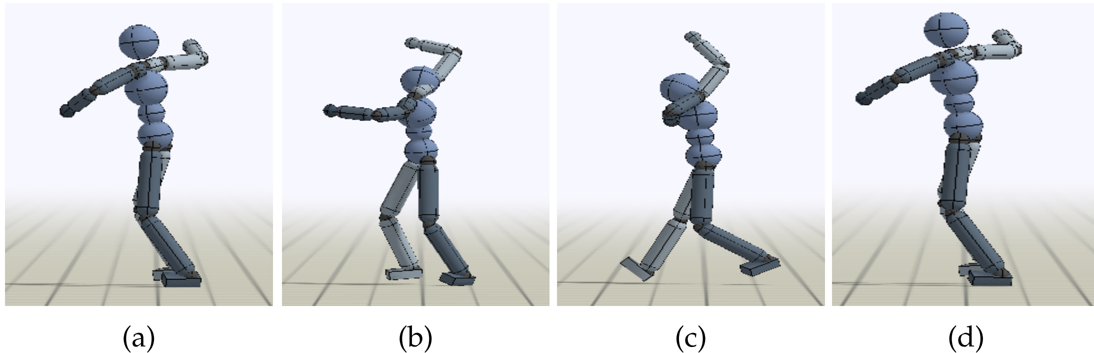
Since the learning code in AMP is written in Python, SWIG was used to create a Python wrapper around `DeepMimicCore` so the learning code could access the functions inside `DeepMimicCore`. The python class in the AMP framework that connects to this SWIG wrapper is `DeepMimicEnv.py`.

6.2 Augmenting DeepMimicCore

Several additions to the codebase were made to `DeepMimicCore`. The major change was to add additional resetting capabilities. The original code of AMP resets to a random time in the motion and matches the reference motion. If the time that was randomly chosen to reset to does not match an exact keyframe, interpolation is done using linear interpolation (lerp) for things such as position, linear velocity, and 1D rotation angles of revolute joints. For quaternions, spherical linear interpolation (slerp) is used instead. For testing purposes and for generating datasets, the ability to manually reset to a specific time is needed. Specifically, the two functions `ResetIndex` and `ResetTime` were added to `DeepMimicCore`. `ResetIndex` allows us to reset the character to a specific keyframe in the reference motion by specifying the index of the keyframe in the JSON file. Similarly, `ResetTime` allows us to reset the character to any time in

the motion. Since the motion is looped, we can specify a time greater than the length of the motion as this will just shift the root of the character.

The reset functions have several parameters that control how much randomness is involved when resetting the character as shown in figure Figure 6.1. With `ResetIndex` and `ResetTime`, random noise can be added to the character's pose and velocity. Additionally, the reset function can sample random angles to add to all joints except the hip and ankle joints, choose whether to also add noise to the velocity of the joints, interpolate between the zero vector and the reference velocity for initialization, etc. The reason why random rotations are not added to the hip and ankle joints is to ensure that if the character is supposed to reset to a standing position, the character won't instantly fall over.



(a) (b) (c) (d)

Figure 6.1: Figure 6.1a shows the character being reset to t = 0 and matching the reference motion like in the original AMP framework. Figure 6.1b resets to t = 0 but each joint (except the hip and ankle) has a random angle added to it. The angle of each added rotation is sampled from $\mathcal{U}(-40, 40)$ degrees and has no knee rotation. Figure 6.1c follows the same procedure as Figure 6.1b but has knee rotation. Figure 6.1d resets to t = 0 but instead adds noise by adding a random noise vector $\mathcal{U}(-0.005, 0.005)$ to the entire state.

For resetting with randomness, the latter method is preferred as the former method of adding a random vector of noise to the state can lead to deformation of the character itself as shown in Figure 6.2.

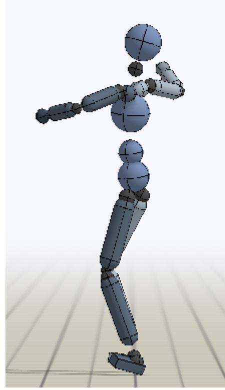


Figure 6.2: Character is reset to t = 0 but a vector of random noise is added to pose and velocity. The noise is sampled from $\mathcal{U}(-0.1, 0.1)$. As we can see, the character becomes disfigured due to this added noise.

For our experiments, we chose to reset to a random time in the motion but not add any additional noise like AMP's reference state initialization as we wanted to get an initial baseline of how MILO would do in the AMP framework.

Most of the other changes to the code were small enhancements such as logging, adding code for getting the expert dataset, or estimating the cost of each state from the dynamic time warping cost which will be discussed in a later section.

6.3 Creating the OpenAI Gym Environment for DeepMimicCore

One of the goals of our research was to create an OpenAI Gym environment for the AMP framework. Specifically, we wanted to create a gym environment that would connect to `DeepMimicCore` in order to allow us test various RL

algorithms.

6.3.1 Gym Environment

OpenAI Gym environments all follow the same API which makes it very easy to create new benchmarks or test various RL algorithms on existing environments and benchmarks. The three main functions in gym environments are:

- `__init__`: The constructor of the gym environment class defines the action and observation spaces for the environment. Additionally, the constructor initializes any necessary variables or objects required for use in the other functions.
- `step`: This function handles executing one step within the environment based on the input action. It returns the next state, the reward, a Boolean flag, `done`, indicating whether the episode has terminated, and a dictionary, `infos`, containing any additional info about the environment.
- `reset`: As the name implies, this function handles resetting the environment and returning the new initial state.

Many environments implement a `render` function for displaying the current state of the environment as well as other helper functions.

6.3.2 DeepMimicGymEnv

To create the gym environment, which we called `DeepMimicGymEnv`, we refactored parts of the code from the DeepMimic and AMP framework in order to

create the required `__init__`, `step`, and `reset` functions. One thing to note is that `DeepMimicGymEnv` is not connected to the RL side of AMP or DeepMimic. Thus, `DeepMimicGymEnv` can be thought of as a Gym environment that wraps around the existing `DeepMimicEnv` class which itself serves as an interface to `DeepMimicCore`.

`__init__`

In the `__init__` function, we define and initialize any variables needed for use in the environment. The first step was to initialize a `DeepMimicEnv` object. We had two possibilities here. The first possibility was to pass in the `DeepMimicEnv` object while the second option was to pass in the necessary arguments and create the object inside `__init__`. With the first option, we quickly ran into issues as doing this would not allow us to use `DeepMimicGymEnv` with multiprocessing libraries. The way Gym environments work with multiprocessing libraries is that, assuming they inherit from `gym.utils.EzPickle`, the multiprocessing procedures (e.g., pipes and pools) pickle the arguments to the constructor of the Gym environment. Then, a new instantiation of the Gym environment is created in each new process using those pickled arguments. The issue is that `DeepMimicEnv` contains the Swig wrapper for `DeepMimicCore` which uses `SwigPyObject` that cannot be pickled. Thus, to use `DeepMimicGymEnv` with multiprocessing, the `__init__` function takes in the arguments required for creating a `DeepMimicEnv` object instead of the object itself.

Besides the arguments needed for initializing `DeepMimicEnv`, the function of `DeepMimicGymEnv` also has parameters for the sampling rate of the policy

(default $\frac{1}{30}$), the update timestep for the world (default $\frac{1}{60}$), the number of substeps for each update of the world (default 10), the seed, and reset arguments used for resetting in the `reset` function. Finally, the observation and action space were defined. Currently, `DeepMimicEnv` only supports the pure imitation scene of AMP or DeepMimic and doesn't support goals but adding this functionality should be a simple extension.

`step`

The `step` function takes in an action and uses this action to step in the world using `DeepMimicEnv`. This code was based off of existing code in `DeepMimic.py` for stepping in the world.

The reward that is returned by `step` depends on how the internal `DeepMimicEnv` object was initialized. If the scene that is loaded is of the `DeepMimic` type rather than `Amp` type, then the reward returned will be the reward $r^G + r^I$ from `DeepMimicCore`. If there is no goal and we are only imitation, then the returned reward is r^I.

In the case where the scene is of the `AMP` type, the returned reward depends on whether the scene is pure imitation or imitation with a goal. In the latter case, the reward returned will be r^G for that timestep computed from `DeepMimicCore`. To get the full AMP reward, we also need r^S but since r^S is computed using the discriminator trained on the RL side, r^S will not be added here since `DeepMimicEnv` does not connect to the RL side of AMP. Thus, `step` will only return r^G.

In the former case where we are only imitating in an `AMP` scene, the reward

returned will always be 0 as $r^G = 0$ and we do not consider r^S. When the episode ends, `DeepMimicCore` actually returns the DTW cost, to be introduced in the next section, as the "reward". Rather than returning this as the reward in `step`, we will place this DTW cost into the `info` dictionary and still return 0 as the reward. The `info` dictionary also contains the Boolean `valid`. If the velocity of the character exceeds a certain threshold, `DeepMimicCore` sets the `valid` flag to false and indicates the trajectory shouldn't be used.

Finally, the dictionary can also be modified to include the state features used in the discriminator. Instead of storing $\phi(s_t)$ by itself, the AMP framework stores these vectors, using the functions `record_amp_obs_agent` and `record_amp_obs_expert`, by storing the concatenation of $\phi(s_t)$ and $\phi(s_{t+1})$ in a "zig-zag" formation. Specifically, the pose portion of $\phi(s_t)$ and $\phi(s_{t+1})$ are concatenated first, followed by the velocity portions. Storing the concatenated vector allows the learning code to store these vectors in one array and not have to worry about concatenation later when training the discriminator. Code was added to get the isolated state feature $\phi(s_t)$ using the function `record_amp_obs_agent_current`. The `step` function in `DeepMimicGymEnv` can be modified to query these state features and store them in the `info` dictionary.

`reset`

The `reset` function uses the reset parameters that were stored, as a dictionary, in the `__init__` function for resetting. The default is to reset to a random time in the motion without adding any extra noise.

`render`

The libraries FreeGLUT, GLEW, and OpenGL are used for visualization. In `DeepMimicCore`, GLEW and OpenGL are used to define the graphics (e.g., character shapes, colors, etc.) while `DeepMimic.py` uses FreeGLUT to manage the OpenGL context. We will focus on the FreeGLUT code in `DeepMimic.py` as we will adapt that for setting up the `render` function. The FreeGLUT code in `DeepMimic.py` works by defining callback functions for updating the screen, keyboard events, mouse events, resizing, etc. Then, `glutMainLoop` is called which enters the GLUT event processing loop. This function never returns and is an infinite loop that responds to events by calling the appropriate callback functions. Thus, in `DeepMimic.py`, several variables are set as global variables such as `RLWorld` object which contains `DeepMimicEnv` and our policy so the callback functions are able to access them. Currently, the display is updated at a framerate of 60fps by utilizing a timer callback function. As the policy updates at a rate of 30Hz, this means that each action corresponds to 2 frames on screen. Additionally, the world updates at a rate of 1.2kHz so each frame shows 20 simulation steps in Bullet.

In order to use FreeGLUT in our environment `DeepMimicGymEnv`, we adapt the code from `DeepMimic.py`. However, we are not able to utilize `glutMainLoop` as the purpose of the `render` function is to display the current state whenever the user desires, so interaction is required. The user will not be able to call the `render` function if we enter the GLUT event processing loop. To get around this situation, we make use of the alternative function `glutMainLoopEvent` which processes one step of the event processing loop. This is typically used inside of an infinite loop along with some other code.

However, we will only call it once in the `render` function so the function is extremely simple. The key lies in the display callback function, `draw`, which is setup using `glutDisplayFunc`. The display function is called using GLUT inside the event processing loop whenever GLUT determines whenever the window needs to be updated. By having `render` call `glutMainLoopEvent`, the idea is that whenever the user calls `step` and updates the state, the next time `render` is called, the `draw` callback function will be triggered, updating the screen.

Unfortunately, since we are not in the infinite event loop, as is typical when calling `glutMainLoop`, we aren't able to use the keyboard and mouse callback functions as well as any other interactive functions. We need to be in the infinite event loop in order to have FreeGLUT respond to these events. By only calling `glutMainLoopEvent`, we are only able to update the display which is sufficient for the `render` function.

We created another file `visualize.py` that mimics `DeepMimic.py` for visualizing our policy. Instead of utilizing the render function of `DeepMimicGymEnv`, `visualize.py` sets up FreeGLUT and uses `glutMainLoop` so we can visualize our policy and interact with the OpenGL window using the keyboard and mouse. Since we still use `DeepMimicGymEnv` for stepping to the next state, the display updates at 30Hz instead of the 60Hz of `DeepMimic.py`. If we wanted to get the 60Hz, we would need to make considerable code changes to `DeepMimic.py`, `rl_world.py`, etc. in order for the AMP framework to load in our policy. While the setup in `visualize.py` doesn't allow us to visualize our policy in 60Hz, the code is considerably simpler and no additional code changes need to be made to the AMP framework.

6.4 Testing MILO

With the `DeepMimicGymEnv` environment created, we chose to test the MILO algorithm. Since there are so many different motions to imitate, we chose to focus only on imitating the spinkick motion without a goal. While this is a looping motion and in theory has an infinite horizon, the training code in the AMP framework sets the horizon at 20 seconds, or 600 timesteps. For our experiments, we set the horizon to be 10 seconds. The following sections will describe the work of this thesis which involved setting up the AMP framework, augmenting it to work with OpenAI gym, and testing the MILO algorithm with the framework. The purpose of this was to test how MILO would perform when the expert dataset only contains states. Since the original MILO algorithm works with (s,a) pairs rather than (s, s') pairs, a few changes were made to the algorithm.

6.4.1 Generating the Offline Dataset

The GitHub repository for AMP contains pre-trained policies for imitating each motion. There are policies that have been trained using the DeepMimic training algorithm as well as the AMP algorithm. Since we are focusing on pure imitation and don't want to deal with manual synchronization using a phase variable, we chose to use AMP-trained policies.

In order to create the offline dataset for training the dynamics model, we needed behavior policies. The policies in the GitHub repository have near-expert performance so they are not well-suited for generating samples for our

offline dataset. The goal was to have an offline dataset with samples from a diverse set of behavior policies with varying performance. Specifically, we aimed for the "average" performance of the policy used for generating the offline dataset to be roughly half that of the expert. In order to generate the behavior policies, we followed the same exact training procedure specified in the AMP paper for training the spinkick policies. Our behavior policies were the various checkpoints that we have saved throughout the training process.

With pure imitation with no additional goals in AMP, the only reward signal is the style reward r^S. Since this reward signal is learned, it does not give us an accurate measure of how "good" our policies are performing until the underlying discriminator is trained sufficiently. Thus, in order to analyze the performance of the behavior policies, we needed to either substitute in a pretrained discriminator or use a different metric. We chose the latter method and decided to use the average pose error as the metric as in AMP. That is, the pose error e_t^{pose} at timestep t is computed between the pose of the simulated character and the reference motion following the equation:

$$e_t^{pose} = \frac{1}{N^{joint}} \sum_{j \in \text{joints}} \|(x_t^j - x_t^{root}) - (\hat{x}_t^j - \hat{x}_t^{root})\|$$

where x_t^j and $\hat{x}_t^j$ are the 3D Cartesian coordinates of joint j from the simulated and reference characters, respectively. Additionally, AMP uses dynamic time warping (DTW) to synchronize the trajectory of the simulated character to the reference motion as the simulated character may perform the action in a slower or faster fashion compared to the reference motion. In our opinion, this method serves as a better metric of performance as it is discriminator-independent and also handles synchronization.

After this cost is computed, a final penalty is applied based on the length

of the horizon. For each "leftover" timestep, one unit of cost is added. To be precise, the penalty added for trajectory τ is equal to $T - len(\tau)$, where T is the horizon. For the spinkick problem, $T = 300$ since we set the horizon at 10 seconds. This cost, the DTW cost, can be thought of a metric between the similarity and synchronization between the simulated character and reference motion while also penalizing trajectories for ending prematurely.

For our dataset, the first step was to collect statistics for each of the checkpoint policies. Specifically, for each checkpoint, we collected many trajectories and computed statistics such as average DTW cost and length along with their respective standard deviations. The average DTW cost and lengths of trajectories are shown in Figure 6.3.

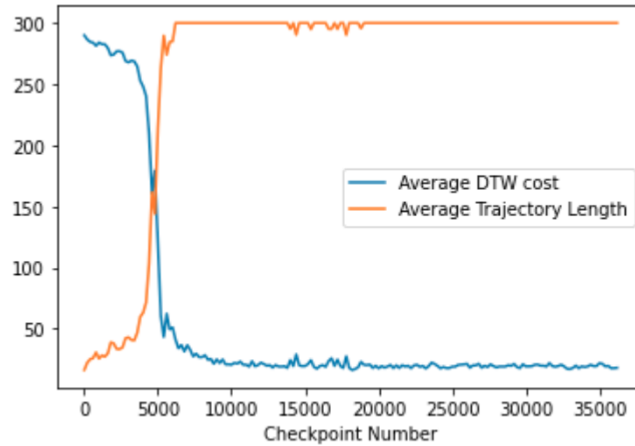
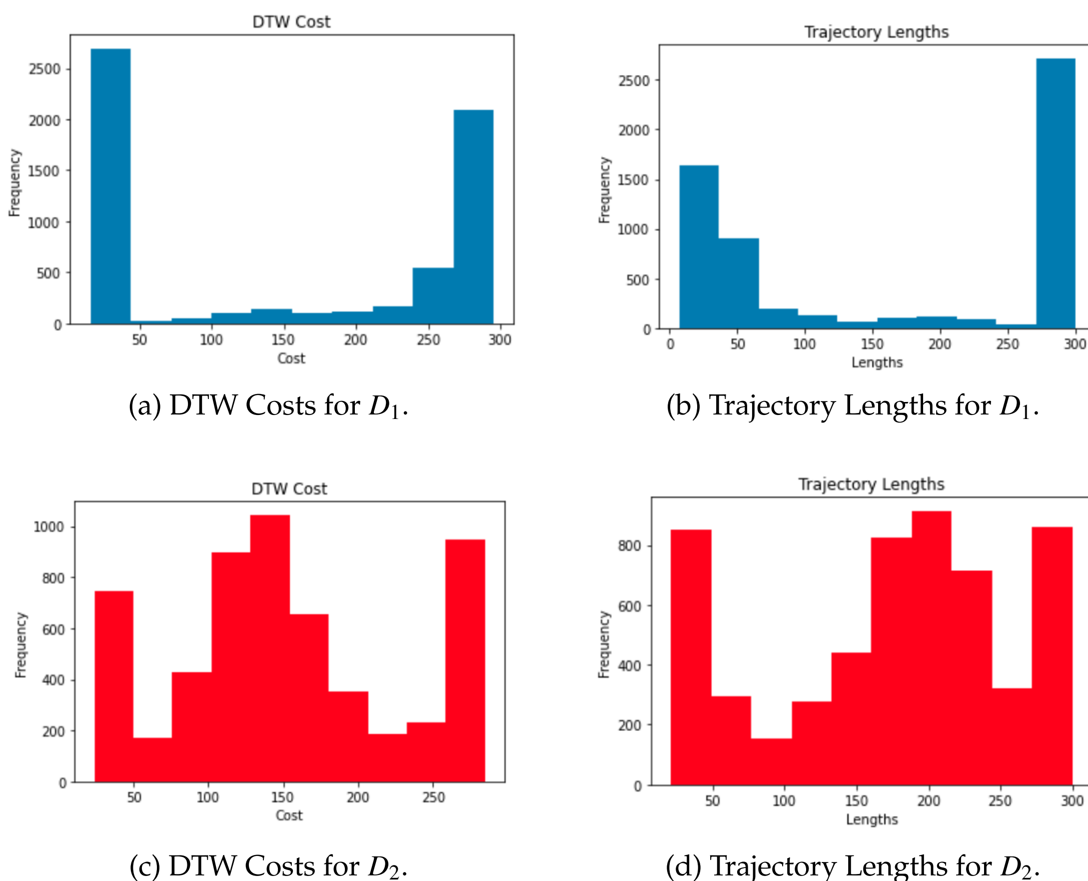


Figure 6.3: Average statistics for various checkpoints collected from training a spinkick character.

As we can see, the average cost and length are roughly inversely proportional which is to be expected given how the DTW cost is defined.

From Figure 6.3, we can see that the worst case DTW cost is 300. Additionally, since our expert in this case is the reference motion which technically has

a DTW cost of 0, we aimed for the offline dataset to have an average DTW cost of ∼150-160 since this would correspond to a dataset with 50% performance of the expert dataset. Using the statistics shown in fig. 6.3, we sampled 1 million samples in order to match the size of the datasets used in the original MILO experiments. Our initial dataset, D_1, utilized samples from all the checkpoints. The statistics for this dataset are shown in Table 6.1 and the average cost is ∼150 as desired but the variance is high as shown by the two-tailed histograms in Figure 6.4a and Figure 6.4b. This was expected given that most of the checkpoint models used performed poorly or performed near expert-level as shown in Figure 6.3.



(a) DTW Costs for D_1.

(b) Trajectory Lengths for D_1.

(c) DTW Costs for D_2.

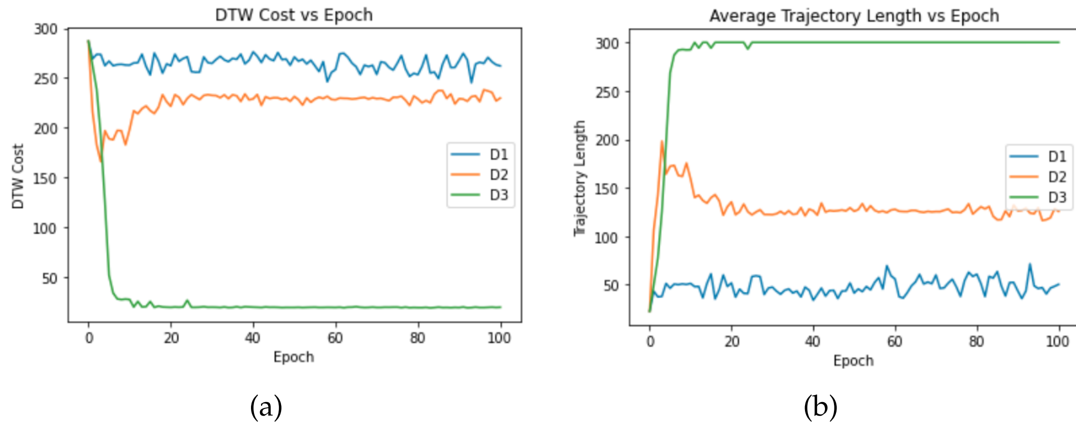
(d) Trajectory Lengths for D_2.

Figure 6.4: The DTW Costs and lengths of trajectories in D_1 and D_2

	Average cost	Std cost	Average Length	Std Length
D_1	149.12	121.39	167.87	126.90
D_2	151.44	76.41	175.14	85.57
D_3	19.58	10.21	299.62	10.53

Table 6.1: Average DTW cost and length of trajectories in D_1, D_2, D_3

In order to reduce the variance in the dataset, we chose to use a smaller subset of policies with most of the policies coming from the checkpoints between iteration 4200 and 5200. In Figure 6.3, this corresponds to the band of policies that didn't perform either near expert-level or very poorly. This dataset, which we will label as D_2, has a much more balanced distribution. To compare these datasets, we trained policies using behavior cloning with D_1, D_2, and a third dataset consisting of only near expert-level policies D_3.



(a) (b)

Figure 6.5: Behavior Cloning results for various datasets.

As we can see in Figure 6.5, behavior cloning with D_2 performs better than D_1. The two-tailed nature of the distribution of trajectories in D_1 makes it so behavior cloning doesn't perform as well as with D_2. The poor performance of D_1 makes sense intuitively as behavior cloning trains the policy to mimic the expert-level behavior policies as well as the extremely poor performing policies

so it shouldn't be able to learn well. Finally, we can see that behavior cloning with a dataset consisting of trajectories collected from expert-level policies results in very good performance.

6.4.2 Training the Dynamics Model

After creating the offline dataset, the next step was to train the dynamics model using D_2. As in the original MILO paper, we trained an ensemble of feed-forward neural networks. We additionally incorporated the suggestion of the authors of MILO to use DenseNet-style [9] feed-forward neural networks. Specifically, the MLP was defined so that each layer obtained the input from all preceding layers, not just its immediate predecessor. Additionally, our models were trained to predict the normalized difference between the next and current state $s_{t+1} - s_t$. In our experiments, we trained several ensembles of four models with various hyperparameters. All ensembles were trained using Adam for 500 epochs and all used the ReLu activation. The hyperparameters that were tuned were the hidden size, number of layers, learning rate and epsilon for Adam, and batch size. We also tested using a scheduler but found that this didn't improve performance by any noticeable margin. The parameters of the best performing model can be seen in Table 6.2.

Table 6.2: Hyperparameters used for dynamic model learning.

Hyperparameter	Value
Hidden Size	(512, 512, 512, 512)
Ensemble Size	4
Activation	ReLU
Optimizer	Adam
Learning Rate	1e-4
Epsilon	1e-4
Batch Size	128

Analyzing the Dynamics Model

The training and validation loss for the dynamics model are shown in Figure 6.6.

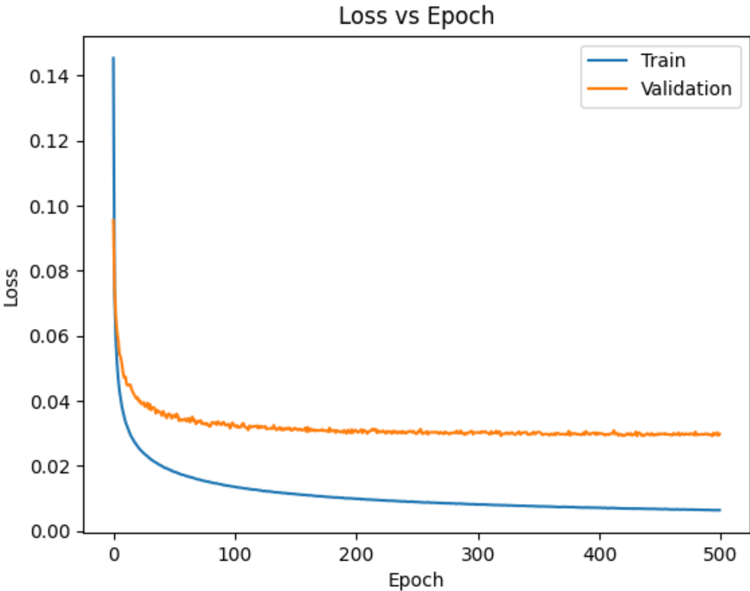
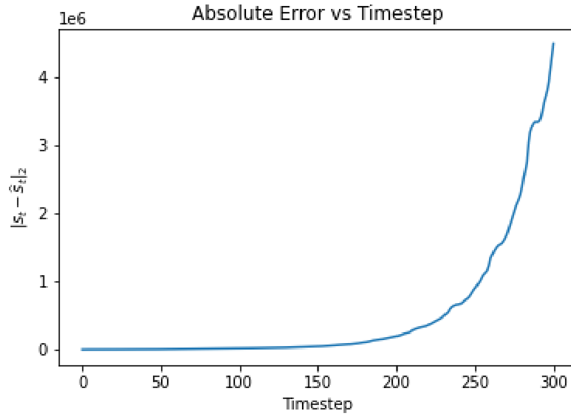
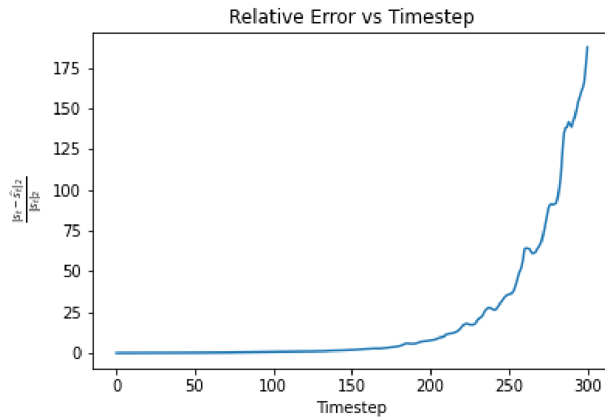


Figure 6.6: Dynamic Model Training and Validation Loss.

Compared to the loss of the dynamics model trained in the original MILO paper, our final loss was much higher. To evaluate how well (or poorly) our dynamics model captured the real dynamics, we replayed several trajectories in our dataset and computed the average relative and absolute error between the reference state and the model state at each timestep. As can be seen in Figure 6.7, the dynamic model quickly diverged from the true dynamics. While it is hard to see on the graph, our relative error surpassed one after $\sim$100 steps and quickly explodes after that point.



(a)



(b)

Figure 6.7: The above graphs show the error in the dynamics model when replaying trajectories. The error at each timestep is computed using the ℓ_2 norm between the current true state s_t and the state in the trajectory rolled out in the dynamics model, $\hat{s}_t$. Figure 6.7a shows the average absolute error $\|s_t - \hat{s}_t\|_2$ at each timestep and Figure 6.7b shows the average relative error $\frac{\|s_t - \hat{s}_t\|_2}{\|s_t\|_2}$ at each timestep.

6.4.3 Generating the Expert Dataset

We used the reference spinkick motions for our expert dataset. In the AMP framework, resetting works by loading the reference motion onto the kinematic character then synchronizing the simulated character to the kinematic motion.

Then, the framework resolves any potential intersections with the ground by shifting the character in the y-axis so that it is in contact with the ground.

To generate the states for the expert dataset, we made use of this pipeline. At each timestep, we loaded the reference motion onto the kinematic character, synchronized the simulated character to the kinematic motion, and queried `DeepMimicCore` the simulated character's state. However, we only resolved any ground intersections at the very start of each trajectory to be consistent with AMP.

The JSON file containing the reference motion for the spinkick contains two trajectories since the keyframes were captured at a frequency of 60Hz but our policy runs at 30Hz. To get the full horizon length trajectory, we looped the motions and to collect more than just two trajectories worth of samples, we reset to a random time in the motion to collect other trajectories. These times usually do not match a keyframe so the keyframes were interpolated to get the states. Our expert dataset consisted of 200 trajectories.

6.4.4 SimEnv - Dynamics Model OpenAI Gym Environment

The next step was to create the OpenAI Gym environment for the dynamics model which we called `SimEnv`. Currently, `SimEnv` assumes that we are simulating an imitation scene with no additional goal and the character we are simulating is the humanoid.

__init__

As usual, the `__init__` function initializes our gym environment. Like in `DeepMimicGymEnv`, `simenv` initializes a `DeepMimicEnv` object as we still need it for resetting purposes. The other key object is the dynamics model which will be used for stepping in this environment. The other parameters include the horizon length, reset arguments, the seed, and `humanoid_json` which contains information about the character. There are also other various parameters such as `transform_veltopos` which specifies whether to store the velocity as the change in position by multiplying by the timestep. More details about these parameters can be found in the documentation.

step

This environment uses the dynamic ensemble for stepping in the environment. This is the key difference compared to `DeepMimicGymEnv`. Since this environment will be used for MILO, the rewards do not matter for the algorithm itself, so the reward is currently hard-coded to return 0 as there is no way to query `DeepMimicCore` to get the rewards.

For determining whether the trajectory has ended, we first take a look at how `DeepMimicCore` determines termination. In `DeepMimicCore`, three different conditions are checked for determining termination of the trajectory. The first is whether the length of the current trajectory has reached the horizon. The second is whether any of the velocities of the `SimLink` objects, which represent the various links of the character, have exceeded a certain threshold. The final condition is whether any parts of the character besides its feet have touched the

ground. We attempt to recreate these three conditions.

The first condition was simple to recreate by simply keeping track of the current timestep using a counter that was updated in each timestep and reset to zero in the `reset` function. Once the counter reached the horizon, we set the `done` flag to true to terminate the trajectory.

To check the second condition, we made use of the fact that the state vector contained the linear and angular velocities of the links. In `DeepMimicCore`, the episode is terminated if the velocity (angular or linear) of any link exceeds a certain threshold. We do the same and set the threshold default to 100 since that is the default in `DeepMimicCore`

The last condition to be recreated was collision detection with the ground. `DeepMimicCore` checks this condition by making use of the Bullet Collision Detection algorithms as each link and joint of the simulated character are Bullet objects themselves. Typically, there is no way to recreate the collision detection as the worlds in AMP typically involve non-flat terrain, objects to avoid, etc. All these entities are Bullet objects, so it is not feasible to recreate collision detection without making the state incredibly large to store all the information required to keep track of everything. The collision detection done by Bullet is highly optimized and more accurate than if we were to attempt to recreate collision detection from scratch.

Luckily, our experiments were done in a world with a flat ground and no other objects for the character to collide with. This made it much easier to check if the character collided with the ground. The first step was to make use of `humanoid3d.txt` which is a character file that contains information on what

kind of Bullet collision shapes were used to represent each link as well as the parameters for each shape. For example, the knees were represented by capsule objects which could be described as a cylinder with half-spheres at each end. For the capsule shape, the character file contains the height of the cylinder as well as the radius of the spheres. Other shapes used for representing links were boxes and spheres.

To check ground collision, we made full use of the state vector. For a given link, we computed the world-coordinate of the center of the link. Then, using the parameters of the shape defined in `humanoid3d.txt` along with the angular rotation of the joint at that point in time, we were able to compute the shape's lowest point. Finally, if the y-coordinate of this lowest point was below 0, we considered the ground collision condition to be met and terminated the trajectory. To assess this method, we used the trajectories in our dataset, $\mathcal{D}_2$, and ran this condition check on each state. We found that our method was accurate as it would usually not tell us the condition was met until the very last states which is desired.

`reset`

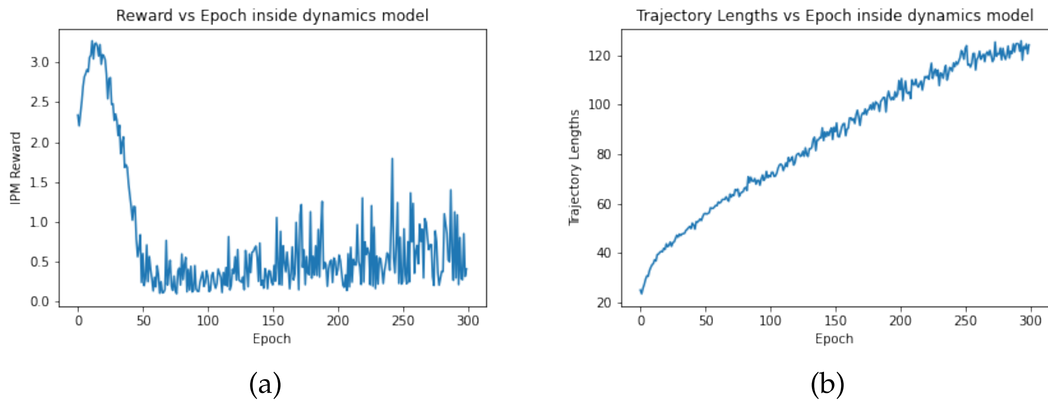
Like in `DeepMimicGymEnv`, the state is reset using `DeepMimicEnv` to a random time in the reference motion with no additional noise added. In addition, we reset the counter keeping track of the length of the trajectory. Finally, we change which dynamics model we use in the ensemble. This is done by keeping a counter tracking how many times we've reset. This counter is used to compute an index (through the modulo operator with the number of models in the ensemble) which selects the model to use in the ensemble for the next trajectory.

6.5 MILO Results

With the offline and expert datasets collected as well as the necessary gym environments created, we tested the MILO algorithm on the AMP framework. As mentioned in the beginning of this chapter, several changes were made to the algorithm. The key change was making the discriminator (s, s') dependent rather than (s, a) dependent as our expert dataset has no actions. Thus, we are now learning from observations only. Due to this, we also had to remove the behavior cloning objective that was added as regularization to the imitation learning objective since behavior cloning is (s,a) dependent. The bonus cost was left as (s, a) dependent, so the final objective became:

$$\underset{\pi \in \Pi}{\operatorname{argmin}} \max_{f \in \mathcal{F}} \left[\underset{\substack{(s,a) \sim d^{\pi}_{\hat{P}} \\ s' \sim \hat{P}(s,a)}}{\mathbb{E}} [f(s, s') + b(s, a)] - \underset{(s,s') \sim D_e}{\mathbb{E}} [f(s, s')] \right] \tag{6.1}$$

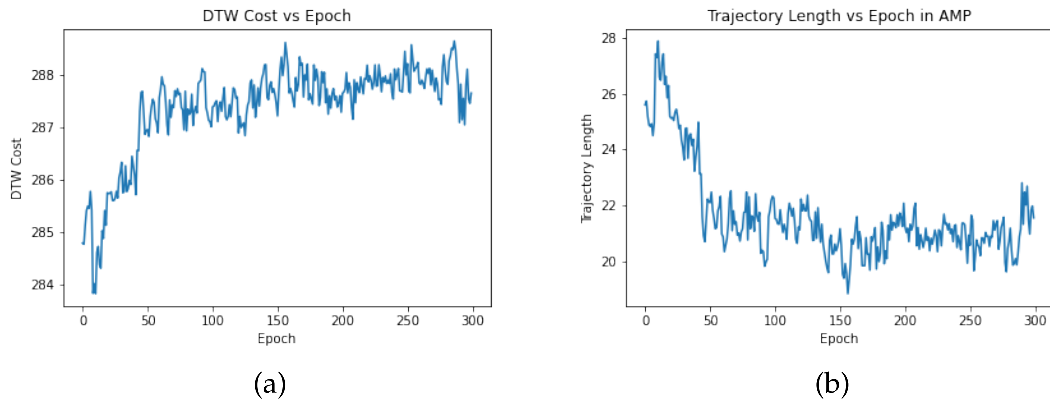
The code from the original MILO and MJRL repositories were adapted to work with the AMP framework. The algorithm used for the imitation learning portion was NPG. The hyperparameters tuned were the size of the actor and critic models, the number of samples used per training, the number of epochs, and the number of epochs for behavior cloning warmstart. For the figures and results below, we used the same hyperparameters as those in the paper. Figure 6.8 shows the average length of trajectories sampled inside the dynamics model as well as the average IPM return. The IPM return is the sum of the IPM rewards, $-f(s, s')$, in a trajectory. While the trajectory length increased, the reward did not increase at all, suggesting that our policy wasn't able to learn.



(a) (b)

Figure 6.8: Figure 6.8a shows the average IPM return which is defined as the sum of the IPM rewards, $-f(s, s')$, for a trajectory. Figure 6.8b shows the lengths of the trajectories inside the model.

Figure 6.9 shows statistics for our policy when tested in the AMP framework. This further confirms that our policy did not learn at all as the DTW cost never decreased and trajectory length stagnated at 20 steps, meaning the simulated character immediately collapsed. We believe the learning failed due to the discriminator not being strong enough to distinguish between expert and generated states. Considering that the discriminator used in AMP was a neural network and was able to perform well, the linear classifier used in MILO most likely was not sufficient for our problem. This is to be explored in future work.



(a) (b)

Figure 6.9: Figure 6.9a shows the average DTW cost for trajectories sampled in the real environment while Figure 6.9b shows the average length of those sampled trajectories.

It is possible that the poorly performing dynamics model also hampered training. We tested using the real dynamics instead of the dynamics model and retrained the policy. However, the results were the same which seems to further suggest that the discriminator was the problem.

CHAPTER 7

FUTURE WORK

The future work can be categorized into code-based future work and experimental future work.

7.1 Code-based Future Work

The main future work for the code lies in extending the functionality of the two gym environments. `DeepMimicEnvGym` was coded specifically for use in our MILO experiments which means that we focused on imitation scenes with no goals. While the code currently does support the addition of goals, it has yet to be evaluated. Additionally, the render function still needs to be implemented.

One major limitation is that `DeepMimicGymEnv` does not connect to the RL side of AMP. As mentioned in section 6.3.2, `DeepMimicGymEnv` relies heavily on the `DeepMimicEnv` object which allows us to interact with `DeepMimicCore`. Without the RL side of AMP, we cannot compute the style reward r^S and include this in the returned reward in the `step` function. To do this, we can simply initialize the `RLWorld` class in `rl_world.py` which actually wraps around `DeepMimicEnv` while also contains all the RL agents used for learning.

For the other environment, `SimEnv`, which contains the dynamic model, the current assumption is that we are imitation a reference motion for a humanoid character with no additional goal. In future work, this environment can be expanded to include goals and other characters besides the humanoid.

7.2 Experimental Future Work

Based on our results in chapter 6, there is a lot to be explored in MILO. As we saw in section 6.4.2, the dynamics model did not perform well so one avenue of future work is figuring out how to improve the dynamics model. Additionally, different methods of collecting the offline data can be explored in future work such as using a metric other than DTW metric to measure performance of policies.

For the imitation learning portion of MILO, we saw in section 6.5 that our policy wasn't able to learn. We believe that the discriminator was not strong enough so future work can explore this area by using different discriminators or using the GAIL-style discriminator used in AMP.

Aside from MILO, various other algorithms can be tested in the AMP framework now that `DeepMimicGymEnv` exists. However, as mentioned earlier, `DeepMimicGymEnv` was only tested on imitation-only objectives without goals so some modifications to the environment may be needed to make it compatible with imitation reference motions with an additional goal.

CHAPTER 8

CONCLUSION

In this thesis, we introduced the AMP framework and adapted it to fit in an OpenAI Gym Environment with `DeepMimicGymEnv`. In doing so, we opened future avenues for applying various RL algorithms on the AMP framework. To demonstrate this, we tested MILO, an offline imitation learning algorithm, using this new gym environment. We illustrated the process used for creating the offline and expert datasets, the dynamics model, and the gym environment `SimEnv`. Finally, we trained the policy using MILO, discussed the results, and highlighted several directions for future work.

BIBLIOGRAPHY

[1] Bullet physics sdk, https://github.com/bulletphysics.

[2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[3] Alekh Agarwal, Sham M. Kakade, Akshay Krishnamurthy, and Wen Sun. FLAMBE: structural complexity and representation learning of low rank mdps. *CoRR*, abs/2006.10814, 2020.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[5] Jonathan D. Chang, Masatoshi Uehara, Dhruv Sreenivas, Rahul Kidambi, and Wen Sun. Mitigating covariate shift in imitation learning via offline data without great coverage. *CoRR*, abs/2106.03207, 2021.

[6] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[7] Arthur Gretton, Karsten M. Borgwardt, Malte J. Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *Journal of Machine Learning Research*, 13(25):723–773, 2012.

[8] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. *CoRR*, abs/1606.03476, 2016.

[9] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[10] Sham M Kakade. A natural policy gradient. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2001.

[11] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[12] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[14] Yongjoon Lee, Kevin Wampler, Gilbert Bernstein, Jovan Popović, and Zoran Popović. Motion fields for interactive character locomotion. *ACM Trans. Graph.*, 29(6), dec 2010.

[15] Sergey Levine, Jack M. Wang, Alexis Haraux, Zoran Popović, and Vladlen Koltun. Continuous character control with low-dimensional embeddings. *ACM Trans. Graph.*, 31(4), jul 2012.

[16] Xudong Mao, Qing Li, Haoran Xie, Raymond Y. K. Lau, and Zhen Wang. Multi-class generative adversarial networks with the L2 loss function. *CoRR*, abs/1611.04076, 2016.

[17] Lars M. Mescheder. On the convergence properties of GAN training. *CoRR*, abs/1801.04406, 2018.

[18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019.

[19] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *CoRR*, abs/1804.02717, 2018.

[20] Xue Bin Peng, Ze Ma, Pieter Abbeel, Sergey Levine, and Angjoo Kanazawa. AMP: adversarial motion priors for stylized physics-based character control. *CoRR*, abs/2104.02180, 2021.

[21] Dean A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 1. Morgan-Kaufmann, 1988.

[22] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007.

[23] Marc H. Raibert and Jessica K. Hodgins. Animation of dynamic legged locomotion. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '91, page 349–358, New York, NY, USA, 1991. Association for Computing Machinery.

[24] Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, John Schulman, Emanuel Todorov, and Sergey Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. *CoRR*, abs/1709.10087, 2017.

[25] Stéphane Ross and J. Andrew Bagnell. Reinforcement and imitation learning via interactive no-regret learning. *CoRR*, abs/1406.5979, 2014.

[26] Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. No-regret reductions for imitation learning and structured prediction. *CoRR*, abs/1011.0686, 2010.

[27] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[28] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[29] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[30] Kihyuk Sohn, David Berthelot, Chun-Liang Li, Zizhao Zhang, Nicholas Carlini, Ekin D. Cubuk, Alex Kurakin, Han Zhang, and Colin Raffel. Fixmatch: Simplifying semi-supervised learning with consistency and confidence. *CoRR*, abs/2001.07685, 2020.

[31] Wen Sun, Sham M. Kakade, Nan Jiang, and Alekh Agarwal. Reinforcement learning: Theory and algorithms.

[32] Wen Sun, Arun Venkatraman, Geoffrey J. Gordon, Byron Boots, and J. Andrew Bagnell. Deeply aggrevated: Differentiable imitation learning for sequential prediction. *CoRR*, abs/1703.01030, 2017.

[33] R. S. Sutton, D. Mcallester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12*, volume 12, pages 1057–1063. MIT Press, 2000.

[34] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1998.

[35] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[36] Faraz Torabi, Garrett Warnell, and Peter Stone. Generative adversarial imitation from observation. *CoRR*, abs/1807.06158, 2018.

[37] S.A. van de Geer and S. van de Geer. *Empirical Processes in M-Estimation*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2000.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[39] Brian D. Ziebart, Andrew Maas, J. Andrew Bagnell, and Anind K. Dey. Maximum entropy inverse reinforcement learning. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, page 1433–1438. AAAI Press, 2008.