KNOWLEDGE TRANSFER FROM EXPERT DEMONSTRATIONS IN CONTINUOUS STATE-ACTION SPACES

by

GABRIEL EWING

Submitted in partial fulfillment of the requirements

for the degree of Master of Science

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

January, 2018

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis of

Gabriel Ewing

candidate for the degree of **Master of Science***.

Committee Chair

Dr. Soumya Ray

Committee Member

Dr. Michael Fu

Committee Member

Dr. Michael Lewicki

Committee Member

Dr. M. Cenk Cavusoglu

Date of Defense

December 5, 2017

*We also certify that written approval has been obtained

for any proprietary material contained therein.

Contents

List of Figures

Knowledge Transfer from Expert Demonstrations in Continuous State-Action Spaces

Abstract

by

GABRIEL EWING

In this thesis, we address the task of reinforcement learning in continuous state and action spaces. Specifically, we consider multi-task reinforcement learning, where a sequence of reinforcement learning tasks have to be solved, and inverse reinforcement learning, where a reward function has to be learned from expert demonstrations. We also use trees to represent models, rewards, and value functions in our domains. First, we design an algorithm to learn from demonstration in the presence of a non-smooth reward function. Second, we design another algorithm to perform offline reinforcement learning in the same scenario. This allows us to re-use experiences to help with new tasks. Third, we introduce a method to incorporate weak knowledge about policies with online learning in policy gradient algorithms. These contributions allow us to create a pipeline that efficiently learns and transfers knowledge across a sequence of tasks. We demonstrate our approaches on the task of learning control of a prosthetic arm from expert demonstrations under various scenarios in simulation.

Chapter 1

Introduction

Imagine that somebody has a prosthetic arm, and they are trying to get through their day. What are the common things they might need to do? They could grab their coffee cup and move it from point A to point B, open a car door, pick up their pen and do some writing, shake hands with a new acquaintance, or a wide range of other tasks.

In a non-prosthetic body part, natural movement is achieved through a combination of muscle memory and fine motor control commands that the brain sends. In a basic prosthetic limb, muscle memory is non-existent and only coarse motor control commands can be sent by the brain. An ideal prosthetic would be able to reproduce the natural movement of an arm as closely as possible with only the coarse motor commands and some limited personalization for each user. In this thesis, we make advances toward such a solution by addressing a variety of challenges that will arise. We now present those challenges, and then a non-technical overview of our attempts to solve them.

1.1 Task Challenges

Consider the problem of just moving between two points in space. In addition to simply moving to the point that the user wants, the arm should follow a natural trajectory, rather than exhibiting the jerky motion often associated with robots. There are several potential benefits to matching the user's expectations as closely as possible: we could reduce the training time required to become comfortable with the new arm, we could prevent the arm from bumping into foreign objects, and we could help the user maintain their balance.

Over millions of years of biological evolution, the movement and perception capabilities of modern animals have become incomprehensibly refined. They allow humans to perform complicated tasks such as walking upright, picking up objects, typing, and much more, without conscious thought. This level of complexity would make it difficult or impossible to define a fixed set of rules to cover all these cases, particularly when compounded with the goal of retaining natural motion.

Another issue is that all of these skills must be *transferable*: we can apply the same basic primitive movement commands to many different scenarios with little additional effort. Walking at a slight gradient takes almost exactly the same directives as walking on flat ground, but our leg and foot positioning changes, different muscles are activated, and our balance must also compensate.

A further compounding challenge is that none of us innately knows how or why we move the way that we do. We tell our bodies to move our hands in some ill-defined flourish, and that command is translated down through various levels of abstraction, none of which we have conscious access to, until it reaches the point of being an actionable impulse for a group of muscle fibers. We could potentially capture these message transactions with neural measuring instruments, but that would be expensive in terms of both effort and money. It would also be vulnerable to the limitations

of modern instrumentation. While we avoid the neural recording issue by using less technically-demanding data, our demonstrations still have to be collected from human subjects. Performing demonstrations is time-consuming and boring, so we need to make the best possible use of the trajectories that we are able to collect. This means that all of our solutions must be data-efficient.

The real world also tends to feature complex, non-smooth decision boundaries: walking safely on the sidewalk and being hit by a bus may be close to each other in Euclidean space, but the quality of outcomes is dramatically different. We need solutions that can adapt to these types of scenarios.

One more major issue is that we act every day in continuous time, space, and actions. Continuous time means that there are no distinct points in time where we are asked to make decisions, but can initiate or alter motions as quickly as our brains can respond to new information. Continuous space denotes that we are not confined to a fixed grid to describe our position. Finally, continuous actions are related to continuous states but mean that we effectively have infinitely fine control over the magnitude and direction of how we move within the constraints of the range of our bodies. While we set aside the continuous-time problem in this thesis, continuous states and actions are essential to a robust solution.

While we focus on the task of arm motion in this thesis, it is merely an instantiation of a more general class of problems. For example, the question of the intention behind movement could potentially be applied to the widely-studied problem of robotic imitation of cockroach movement [1]. Another relevant scenario is the self-driving car problem, which could incorporate multiple pieces of our work: reproducing human intention in driving, transfer of knowledge between different driving conditions, and choosing actions in continuous space. We have made an effort to ensure that our algorithms are capable of generalizing to these other problems with only minimal additional work.

Challenge	Algorithmic approach/Contribution	Chapter
Learning from demonstration	Tree-structured inverse reinforcement learning with continuous actions	4
Synthesizing previous knowledge into a transferable component	Fitted Q-Iteration with continuous actions	5
Identifying similarities to previous tasks	Model Reuse	6
Combining transferred knowledge with online learning	Model-augmented deterministic policy gradient	6

Table 1.1: Task challenges and associated responses

1.2 Algorithmic Solutions

All of these considerations suggest that explicitly programming movement commands for a prosthetic might not be the most efficient approach. One alternative, a data-driven methodology that we explore in this thesis, is to instead learn from recordings that we make of people's arm movements when they move between a given set of points. Rather than just trying to perfectly reproduce these trajectories with a prosthetic, we use them to find, at one level of abstraction, *why* the arm follows the trajectories.

It is important to note here that we are not incorporating any biological modeling of the activated muscles and neurons. Instead, we are representing the arm as a computational process and trying to recover an associated intention that would have led to the arm following the paths that it did.

If we can find an accurate intention, then we can create our own process with that same intention that gives rise to similar behavior. We are coming up with an abstracted function that allows us to compute similar paths, without a physiological

explanation for why the arm demonstration followed a trajectory.

Once we have an answer for the intent behind the movements, we can build a system that learns to match that intent. This offers several advantages: it generalizes better than simple imitation would to motions that we didn't record, it allows us to better address the problem of transfer, and it could give us some insight into the mechanics of a non-prosthetic arm.

Our approach to the transfer problem relies on *model reuse*. For our purposes, this is the ability to recognize when a new situation is similar to one that we have seen previously. If we can accurately identify similarities between tasks, then we can use the knowledge that we gained during a previous task to jump-start learning for the new one.

We address non-smooth feedback by using tree structures. Many machine learning algorithms make the assumption that two points that are close to each other in space should have similar outcomes. This is helpful in many scenarios, but for certain situations it is not desirable, such as those with discontinuous utility boundaries. Trees are able to handle both: they can generalize or create sharp distinctions as appropriate.

The problem of continuous states and actions is a recurring theme in this thesis. Since the most interesting existing algorithms in multiple stages of our pipeline deal with discrete actions, we have to find suitably altered or new approaches.

1.3 Overview

The remainder of this thesis is structured as follows. Chapter 2 gives more information about the data collection and prior work on related issues. Chapter 3 describes in detail the machine learning pipeline that we build and how each piece fits together, as well as the concrete environments that we use for our experiments. Chapter 4 covers

the first step of the process, learning the intentions of the movements we recorded. Chapters 5 and 6 address how we use the movement intentions to generate our own movements.

Chapters 4, 5, and 6 all have empirical evaluations of the contributions described in those chapters. In Chapter 7, we present experiments on our full multi-task pipeline, combining the pieces from each of the previous three chapters. Finally, Chapter 8 reviews our contributions and gives some avenues for possible future work.

Chapter 2

Background and Related Work

In this chapter, we first give a brief introduction to the problem space (Sections 2.1-2.5), including some of the foundational algorithms in the reinforcement learning field. Then, we present the more recent algorithms that we modify and build upon for our contributions (Section 2.7).

2.1 Markov Decision Processes

In this thesis, we are concerned with sequential decision-making problems (SDMs), where an agent is presented with a choice to make at multiple time points. The standard way of modeling SDMs is in the framework of Markov Decision Processes (MDPs). An MDP is defined by a tuple $(S, S_0, A, P, R, \gamma)$. S is the set of states that the environment can be in at a point in time. Unless otherwise stated, we assume that a full representation of the state is available to the agent in an MDP. That is, the agent has access to all the state features that it needs to choose optimal actions. $S_0 \subset S$ is the set of possible starting states for the MDP. A is the set of actions available to the agent at each step. This can be a finite set such as $\{UP, DOWN, LEFT, RIGHT\}$ or an infinite set such as $[-1, 1]$. $P : S \times A \times S \rightarrow [0, 1]$ is the transition function, giving the probability that action A chosen in a given state will lead to a specific outcome

state. $R : S \times A \rightarrow \mathbb{R}$ is the reward function, which is the value that the agent tries to optimize through actions. Finally, $\gamma \in [0, 1]$ is the discount factor, giving the amount that the agent should prefer rewards received immediately to those received at future time steps. With $\gamma = 0$, the agent only cares about immediate rewards, and with $\gamma = 1$, it gives no preference to immediate rewards instead of later rewards. For $0 < \gamma < 1$, it still considers future rewards but gives more weight to those received immediately.

Given an MDP, we consider the notion of a policy, $\pi : S \times A \rightarrow [0, 1]$, where $\pi(s, a)$ represents the likelihood of taking action a in state s, and $\int_a \pi(s, a) = 1$ for any s. A stochastic policy can have $\pi(s, a) > 0$ for multiple different a in state s, whereas a deterministic policy has $\pi(s, a) = 1$ for some a in s. Generally, agents try to find an optimal policy π^*.

Optimality is determined by the rewards received during execution of a policy, weighted by the discount factor. We can define the cumulative discounted return, or utility, of a trajectory T of length n. T consists of the states, actions, and rewards seen during some execution of an MDP while following a policy π. The utility $U(T)$ is calculated as follows:

$$U(T) = r_0 + \gamma r_1 + \gamma^2 r_2 + ... + \gamma^{n-1} r_{n-1} \tag{2.1}$$

In a deterministic environment, it would be sufficient for the agent to try to maximize $U(T)$ in an optimal policy. However, most interesting MDPs have stochastic transitions, and can have multiple starting states. Therefore, we need to take into account the expectation of the discounted rewards over all trajectories. The cumulative discounted expected reward function, $J(\pi)$, is:

$$J(\pi) = E[U(T|\pi)] \tag{2.2}$$

Our goal is to maximize $J(\pi)$ by finding the optimal policy π^*:

$$\pi^* = \arg\max_{\pi} J(\pi)] = \arg\max_{\pi} E\left[\sum_{t=t_0}^{\infty} \gamma^t r_t | \pi\right] \tag{2.3}$$

2.2 Value Functions

We defined the optimal policy π^* in the previous section using the notion of the expected discounted reward received over the entire MDP. It is useful to define the expected discounted reward received under some policy π at a single given state s. We refer to this as the value function, $V^\pi(s)$. $V^\pi(s)$ takes into account not just the reward received by following π at s, but the expected reward received under π at all future states as well. We are able to calculate this using a form of the dynamic-programming Bellman equation [2]:

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s \in s'} P(s, \pi(s), s') V^\pi(s') \tag{2.4}$$

The Bellman equation shows us that the value function can be decomposed into two pieces: the immediate reward received ($R(s, \pi(s))$), and the expected future discounted reward ($\gamma \sum_{s \in s'} P(s, \pi(s), s') V^\pi(s')$). By encapsulating these two components in the single term $V^\pi(s)$, we can reason about the utility effects of policy changes without concerning ourselves with when the effects occur.

2.2.1 Action-Value Function

The value function $V^\pi(s)$ is a useful descriptive metric for the utility of a policy at a given state. It is also useful to establish the idea of the utility of a given (state, action) pair. We call this the action-value function, $Q(s, a)$. We can calculate the

action-value function using a different form of the Bellman equation:

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s \in s'} P(s,a,s') Q^{\pi}(s', \pi(s')) \tag{2.5}$$

Unlike the value function, the Q-function allows us to choose between different actions at a state by telling us which has the best expected discounted utility. This is powerful because it can go beyond a descriptive metric and inform the actual choices that we make in an MDP. Ideally, the policy π that is followed in state s' and beyond will be the optimal policy, π^*. We can similarly define the optimal action-value function with another form of the Bellman equation:

$$Q^{*}(s,a) = R(s,a) + \gamma \sum_{s \in s'} P(s,a,s') \max_{a' \in A} Q^{*}(s', a') \tag{2.6}$$

If we knew $Q^*(s,a)$ for all (s,a) pairs of an MDP, then most of the work in this thesis would be irrelevant because we could trivially act to optimize the expected utility. Unfortunately, this is not the case, and there is a significant amount of prior work on various ways to calculate or approximate the Q-function. We will revisit this topic in Section 2.4.2.

2.3 Basis Functions

In smaller MDPs, we can represent a policy or Q-function by enumerating each state or (state, action) pair and storing the corresponding action or Q-value in a table. This is clearly an optimal representation if it is feasible: we would have the exact value of each entry without any signal loss. However, creating and storing a full enumeration is often expensive or impossible. An MDP with 100,000 states and 1,000 actions per state would take 100,000,000 table entries to represent the Q-function, which would be possible to store with modern hardware. The problem would be in learning those

Q-values: we would need to experience each of those 100,000,000 transitions at least once to learn a reasonable value, and, as we will see in Section 2.4.2, potentially many more times than that. Furthermore, in a continuous-state or continuous-action MDP, there are infinite possible (state, action) pairs and tabular representations are impossible.

Therefore, for most MDPs, we employ alternative representations using basis functions. A state-action basis function ϕ performs the following mapping (it is also possible to parameterize ϕ with only a state):

$$\phi(s,a) = \begin{bmatrix} \phi_0(s,a) \\ \phi_1(s,a) \\ \vdots \\ \phi_n(s,a) \end{bmatrix} \tag{2.7}$$

where each element ϕ_i of the output vector is a different numerical value, corresponding to some information extracted from the state and action variables. The simplest basis functions are just a direct correspondence to the variables observed by the agent. In a game of soccer, the state features might be the positions and velocities of all players and the ball, and the actions features the direction and force with which a dribbling player could kick the ball.

The basis function can also extract useful information from the observations and include that as well: in the same soccer example, one such piece of information might be the number of defending players between the ball and the goal.

Additionally, in some MDPs, it is useful to consider a discretized, tiled representation of the state space. In a tiled representation, a continuous state variable is assigned to a number of tiles, or bins. A bin is responsible for a continuous subset of the possible values of that variable, and the disjoint union of the subsets of all bins covers all of the possible values of the variable. Each bin corresponds to a different

basis entry ϕ_i. This means that for a given variable value, the entries of the basis vector for that variable's bins will have one value of 1 and the rest 0. We can also make bins out of combinations of variables: a single bin corresponding to a range of one variable and a range of another. Taken to the extreme, we might combine all state variables into bins, so that only a single entry in the entire basis vector is 1 and the rest 0.

Basis functions alone are insufficient to represent a policy or Q-value, because they only encode information about the state and actions, not their quality. To make use of them, we need a corresponding vector of weights, w. Each weight w_i maps to a different entry in the basis vector, ϕ_i. Given w, we could calculate the Q-value of an (s, a) pair by the dot product $Q(s, a) = w \cdot \phi(s, a)$. Learning appropriate values for the w_i is the central problem in learning a good policy or value function.

2.4 Reinforcement Learning

Reinforcement Learning (RL) is the field of machine learning dedicated to autonomously learning optimal policies for MDPs. Most RL algorithms fall into one of two categories: online, or offline. Our contributions in this thesis concern both online and offline RL, and our main RL strategy is a modified online algorithm that incorporates offline learning algorithms.

2.4.1 Online RL

Online RL has the agent learn a policy while taking actions in the environment. While the policy is usually bad at first, achieving a low cumulative return, standard online RL algorithms improve the policy through updates from the rewards received and are generally able to converge to good policies on a wide variety of problems.

A standard conflict in online RL is the trade-off between exploration and exploita-

tion. Exploration is the need for the agent to learn new things about the environment by seeing new states and trying new actions. If the agent acted according to its current notion of the best policy at all times, it may never discover better possibilities. Exploitation is the desire to act to make use of the knowledge that the agent already has and optimize the expected utility of its actions. A successful online RL algorithm and policy must balance these two aspects.

One common, naïve form of exploration is ϵ-greedy exploration, where the agent chooses the action that it currently thinks is best most of the time, but with some probability ϵ chooses uniformly from the other possible actions. Another is to structure the policy as a probability distribution. With a Gaussian policy, the agent would update the mean of the distribution but actions would be chosen according to a Gaussian distribution around that mean. The variance can be either fixed or another parameter to be learned.

Most RL algorithms maintain some policy π parameterized by a vector of weights, w, and update w based on their experiences. The learning rate, α, is the amount that each update is allowed to alter the policy at one time. This prevents past updates from having their contributions entirely overwritten from more recent inputs.

2.4.2 Q-Learning

In Section 2.2.1, we introduced the notion of the action-value function, $Q(s, a)$. One of the most common approaches to the RL problem is to learn a good approximation of the Q-function, and then create a policy by selecting the action with the highest Q-value in each state. This is known as Q-learning.

As discussed in Section 2.3, to perform Q-learning with a basis $\phi(s, a)$, we maintain and update a weight vector w. The Q-value for an (s, a) pair is calculated by $Q(s, a) = w \cdot \phi(s, a)$. At the beginning of learning, we generally have no knowledge as to what good values for w are, so we initialize w to the zero vector or some small random

weights.

To update w, we return to the Bellman equation from Section 2.2.1. For some (s, a, s', r) transition, we calculate the new Q-value:

$$Q_{n+1}(s, a) \leftarrow r + \gamma \max_{a \in A} Q_n(s', a) \tag{2.8}$$

and then move w in the direction necessary to produce that new Q-value. We discount our update by the learning rate α so that the policy does not change too suddenly.

Q-learning works well on many MDPs, assuming that it has access to a good basis function ϕ. Its large drawback from our perspective is that it has to iterate through all possible actions to find the best Q-value in the next state. This makes Q-learning slow for MDPs with a large number of discrete actions, and impossible for MDPs with continuous actions unless we restrict the policy to a finite subset of those actions.

2.4.3 Policy Gradient

Policy gradient algorithms [3] take a different approach. As their name suggests, they rely on the differentiability of the expected discounted cumulative reward $J(\theta)$ with respect to the policy parameters θ. The main idea is to move the policy parameters in the direction of that gradient to improve $J(\theta)$.

Sugiyama [4] gives the gradient update for the policy parameters θ, given a sequence of trajectories N, each trajectory consisting of T (s, a, r) transitions, as:

$$\nabla_\theta \hat{J}(\theta) = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi(a_{t,n} | s_{t,n}, \theta) r_{t,n} \tag{2.9}$$

At each step, this gradient is multiplied by the learning rate and added to the policy parameters. The calculation of the gradient of π depends on the specific form of π. For a Gaussian policy, for example, that draws actions a from a distribution parameterized by state s, mean μ, and standard deviation σ, the update for θ is split

into the updates for μ and σ. The equation for the μ update is (from [4]):

$$\nabla_\mu \log \pi(a|s, \mu, \sigma) = \frac{a - \mu^\intercal \phi(s)}{\sigma^2} \phi(s) \tag{2.10}$$

Intuitively, policy gradient on a stochastic policy works by evaluating which of the available actions lead to the highest expected discounted cumulative reward, and then altering the policy to perform those actions more. We will re-visit this topic to discuss policy gradients with a deterministic policy in Section 2.7.1.

2.4.4 Offline RL

In offline RL, there is no direct interaction with the environment. Instead, the learner is given a set of transitions sampled from the environment via some other policy and tries to find the best policy using that knowledge. This has some important advantages and disadvantages versus online RL. One advantage is that taking actions in the environment is often costly, and offline RL requires no further sampling expenditure once the samples have been collected. One disadvantage is that offline RL usually cannot choose to explore certain regions of the state and action space further, while online RL agents can. We now briefly give an overview of one popular family of offline RL algorithms that is foundational to our offline RL work.

2.4.5 LSPI

Policy iteration starts with an arbitrary policy and gradually shifts it toward optimality. It does this by interleaving steps of *policy improvement* with steps of *policy evaluation*.

Lagoudakis and Parr [5] introduced the Least-Squares Policy Iteration (LSPI) algorithm for MDPs with large or continuous factorizable state spaces and discrete action spaces. Algorithm 2.1 is an instantiation of LSPI.

The main loop of the algorithm has three steps. First, for each (s, a, s', r) transition, it updates the Q-value of the pre-transition (s, a) pair using the Bellman equation on r and $Q^\pi(s')$, which is our current estimate of the best Q-value that can be achieved in state s'. This is the policy evaluation step. Then it does a least-squares regression from the basis function ϕ and weights w to the updated Q-values. This gives us a generalized, functional form for Q as opposed to the table generated by the first step. Finally, it updates the $Q^\pi(s')$ using the new w. This is the policy improvement step. LSPI's generalization capability is limited by the use of a linear function

Algorithm 2.1 LSPI

Input: Samples $(s, a, s', r)_{1..n}$, Basis function $\phi(s, a)$, Number of iterations j, Discount factor γ
Output: Policy $\pi(s)$

$Q_0^\pi(s')_{1..n} \leftarrow 0$
for all $k \in 1..j$ **do**
  $Q_k(s, a)_{1..n} \leftarrow r_{1..n} + \gamma Q_{k-1}^\pi(s')$ // Policy evaluation
  $w_k \leftarrow \arg\min_w \sum_{i=1}^{n} (w \cdot \phi(s_i, a_i) - Q_k(s_i, a_i))^2$ // Least-squares regression
  $Q_k^\pi(s')_{1..n} \leftarrow \max_{a' \in A} w_k \cdot \phi(s', a')$ // Policy improvement
end for
$\pi(s)_{1..n} \leftarrow \arg\max_{a \in A} w_k \cdot \phi(s, a)$
return $\pi(s)_{1..n}$

for the Q-values. It still can find good policies on many MDPs given a friendly basis function ϕ. However, creating such a basis function can be challenging and requires human intervention. We would like to be able to do policy iteration with a simple basis function and still be confident of learning good policies. This leads us to the Fitted Q-Iteration algorithm, which we will discuss in Section 5.1.

2.4.6 Model-Based RL

In reinforcement learning, a model is any additional information about the environment (either learned or given *a priori*) apart from the policy or value functions.

Usually, models attempt to represent either the transition dynamics or reward function of the environment. If the agent had access to a perfect model of the transition dynamics and reward function, it should be able to find an optimal policy for the environment. One way to do this would be to generate a large number of transition samples, covering the entire state and action space, and run an offline RL algorithm on those samples. In practice, models are usually imperfect and other methods must be used.

Model-based algorithms have the advantage of generally being able to derive more information—and therefore better policies—from fewer experiences than model-free algorithms, at the cost of computational time and storage space. All of the RL algorithms that we have seen so far in this thesis have been model-free algorithms.

In Chapter 6, we will present a novel RL algorithm that incorporates the use of models into the normally model-free family of policy gradient algorithms.

2.5 Learning from Demonstration

The problem setup of learning from demonstration is as follows: we are given a number of trajectories ("demonstrations") of some agent (the "expert") performing a task. We want to learn to perform the task from the trajectories. There are two general approaches to the learning from demonstration: imitation learning, and inverse reinforcement learning (IRL). In imitation learning, the idea is that we should match as closely as possible the expert's policy; that is, faced with a decision between multiple actions, we should choose the action that we think the expert would have chosen. While imitation learning is good at mimicking the expert, it does not give us as much insight into the problem and is less robust to changes in the environment or task than IRL.

2.5.1 Inverse Reinforcement Learning

IRL, rather than immediately trying to recover the expert's policy, first focuses on learning the reward function that the expert was implicitly exploiting when performing the demonstration. After we have the reward function, we can do forward RL in either the real environment or a simulator, using the IRL reward output to generate synthetic reward values, until we have a policy to optimize the IRL reward. IRL is attractive because the reward function is useful in transfer to problems with altered dynamics, and because having a numerical approximation of the reward function can tell us more about the problem domain itself and why the expert chose its policy.

One challenge in IRL is the indistinguishability of the true reward function. Even with very extensive expert demonstrations, there are still an infinity of reward functions on which the expert's actions are optimal. Foremost among these is the null reward: if the expert simply receives a reward of zero in every state, then any policy is optimal. While this is an interesting issue that is open to further exploration, we acknowledge its existence but set it aside for the most part in this thesis.

2.6 Supervised Learning

The broad focus of this thesis is on RL, which can be briefly summarized as learning to perform tasks that require decisions to be made in a sequential manner. Many RL algorithms, though, including ours, incorporate algorithms from another branch of machine learning: supervised learning. In supervised learning, the goal is to learn to predict labels from some set of features. For example, a supervised learning algorithm could be trained to predict a person's preferred operating system based on their occupation, age, income, and other factors.

Supervised learning can further by divided into two different categories: classification, and regression. In classification, the labels generally belong to a small finite set

of possible values. The operating system example above would fit in this category: the set of labels would be Windows, MacOS, GNU/Linux, various BSDs, etc.

In regression, the labels belong to a continuous range of values. An example would be the task of predicting a dog's weight in kilograms based on its breed and age. In this thesis, we focus more on regressors, due to the continuous-action nature of our work and the continuous nature of reinforcement learning concepts such as Q-values.

2.6.1 Tree Models

One way to partition the output space in both classification and regression is through tree-structured models.

Decision trees are classifiers, with each leaf corresponding to a class label. The internal nodes of the tree are tests on the input features, and the children of a node correspond to the outcome of that test. To classify a new example, the test at the root node is applied, the appropriate child is selected, and so on, until a leaf is reached. The output is the label of that leaf.

Each node has a single feature that it uses to make a decision about an example. Based on the value of that feature, the node will decide which of its children it should choose to send the example to. Each child corresponds to a subset of the range of possible values that the node's feature can have.

The leaf nodes (nodes that have no children) do not have single-feature tests. Instead, they make the final decision about what the example's predicted label should be. The simplest way to do this is to have a fixed prediction of the most common value of all labels of the examples that created the leaf when the tree was trained.

Regression trees are similar to decision trees, but perform regression rather than classification. This can be achieved through a variety of different changes to decision trees, including the CART algorithm [6]. In this thesis we primarily use model trees, introduced by Quinlan in 1992 [7].

2.6.2 Tree Construction

Once a tree exists, using it to make predictions is easy. Most of the complexity and difficulty with tree models comes in creating the tree.

Algorithm 2.2 shows the generic tree creation process. We start with some training set consisting of examples and labels. Each example has some set of features. At each level of the tree, we need to choose one feature (Step 4), and one particular value of that feature (Step 5), upon which to make the node split. Once we choose a feature and value, we partition the training data on that test and train each child node with the appropriate subset of the data. We repeat this process until the criteria we have established for stopping the splits and creating a leaf becomes true (Step 1).

The primary question for any training algorithm is how it selects the feature and value to split on. Ideally, the split decision should be consistent in some way with the method of labeling a leaf, to produce the best leaves possible.

Algorithm 2.2 CONSTRUCT-TREE

Input: Examples $X_{1..n}^{f_1..f_k}$, labels $Y_{1..n}$
Output: Root node of final tree
1: **if** SHOULD-MAKE-LEAF(X, Y) **then**
2: **return** MAKE-LEAF(X, Y)
3: **end if**
4: **for all** $f \in f_1..f_k$ **do**
5: **for all** $i \in 1..n$ **do**
6: $P_i^f \leftarrow$ SPLIT-QUALITY(X_i^f, X, Y)
7: **end for**
8: **end for**
9: $f^*, v^* \leftarrow \arg\max_f \arg\max_x P_x^f$ // feature and value of the best split
10: $(X, Y)_{1..m} \leftarrow$ PARTITION(X, Y, f^*, v^*) // partition the examples and labels based on the best split
11: $children \leftarrow \varnothing$
12: **for all** $X_j, Y_j \in 1..m$ **do**
13: $children \leftarrow children \cup$ CONSTRUCT-TREE(X_j, Y_j)
14: **end for**
15: $root \leftarrow$ NODE$(f^*, v^*, children)$
16: **return** $root$

Three functions called in Algorithm 2.2 require non-trivial implementations: SPLIT-QUALITY, SHOULD-MAKE-LEAF, and MAKE-LEAF. We will return to these functions for our various tree implementations in later chapters.

Note that the call to PARTITION may result in either a binary or multi-way split, depending on the other implementation details.

In this thesis, we frequently make ternary splits, so there is one child that is assigned samples less than the split value, one child with samples equal to the split value, and one greater. If the split value is at one extreme end of the value range, there is no data to create either the $<$ or $>$ child, so for tree induction we assign any such samples to also go to the equal child.

2.7 Related Work

The previous sections in this chapter gave basic background about the problem spaces that we are addressing. In this section, we present the algorithms most closely related to our work. In some cases, we directly alter or build upon them in later chapters to make them applicable for our goals.

2.7.1 Deterministic Policy Gradient

We discussed the family of policy gradient algorithms in Section 2.4.3. Prior to 2014, all published policy gradient algorithms only worked for stochastic policies. That changed with Silver et. al's "Deterministic Policy Gradient Algorithms" [8].

Deterministic Policy Gradient is based on an actor-critic algorithm. Actor-critic algorithms exploit the connection between the gradient of cumulative discounted expected return, $\nabla_\theta J(\theta)$ and Q-values. The "actor" uses an estimate of the Q-values to compute an approximation of $\nabla_\theta J(\theta)$ and correspondingly update the policy parameters θ, and the "critic" updates the estimate of the Q-values. However, finding

$\nabla_\theta J(\theta)$ requires the ability to take $\nabla_\theta \log \pi_\theta(a|s)$, which is impossible for deterministic policies because most of the support of $\pi_\theta(a|s)$ is zero, and undefined under logs. This means that most policy gradient algorithms, and by extension actor-critic algorithms, use stochastic policies.

Deterministic Policy Gradient is enabled by using an alternate formulation of the gradient update, avoiding logs. They give multiple variations of algorithms that implement this formulation, including both on-policy and off-policy algorithms. The one that we use follows an off-policy algorithm for exploration: that is, the policy used to collect samples during the learning procedure is not the same one as is returned for evaluation purposes.

First, we calculate the Bellman difference for the critic weights w:

$$\delta_k \leftarrow r_k + \gamma Q^w(s_{k+1}, \mu_\theta(s_{k+1})) - Q^w(s_k, a_k) \tag{2.11}$$

Then, we use the Bellman difference to update the critic weights:

$$w_{k+1} \leftarrow w_k + \alpha_w \delta_k \phi(s_k, a_k) \tag{2.12}$$

Finally, the critic weights are used to approximate the policy gradient:

$$\theta_{k+1} \leftarrow \theta_k + \alpha_\theta \nabla_\theta \mu_\theta(s_k)(\nabla_\theta \mu_\theta(s_k)^\top w_k) \tag{2.13}$$

The learning rates, α_w and α_θ, can be configured separately from each other.

While our algorithms are not directly tied to Deterministic Policy Gradient, we use it for many experiments and present a modified version in Section 6.6.1.

2.7.2 Arm Prosthetic Work

Machine learning with arm prostheses has been explored in the past before. Not included in this section is the prior prosthetic work that we directly build on, which we will instead discuss in Section 3.3.

Thomas, in [9], applied a continuous-time actor-critic algorithm to forward RL for a prosthetic. However, they did not investigate the problem of learning from demonstration or the problem of knowledge transfer.

2.7.3 Policy Reuse

Policy reuse, explored in [10] and [11], among others, involves using past policies to improve exploration or the rate of learning on new domains. In [10], they assumed that a library of past policies was provided and used them to bias exploration. They did this by creating a similarity metric between the past policies and the current environment, calculated from the performance of the past policies when run on the current environment. Our approach, by contrast, involves creating models of previous environments. This allows us to define our own similarity score, which we can compute without altering the exploratory policy. We discuss this further in Chapter 6.

2.7.4 Bayesian RL for the Multi-Task Setting

Multi-task reinforcement learning (MTRL), the main paradigm that we address, has been extensively explored in the literature.

Dearden et al. [12] used a Bayesian posterior distribution over environment model parameters to guide exploration. They used the parameter uncertainties to create distributions over Q-values.

Wilson et al. [13] built on this work to create and update a distribution on past MDPs with a hierarchical Bayesian model. This allowed them to quickly learn about

new environments if they were similar to a past environment, or avoid using past knowledge if it was not suitable.

Our approach in Chpater 6 can be viewed as a weaker version of the Bayesian approach.

Chapter 3

Overview of Contributions

In this chapter, we first present the general structure of the problems that we are trying to solve. Then we give a high-level description of how the various parts of our contributions fit together. The third section contains the concrete details of the domains that we evaluate our algorithms on.

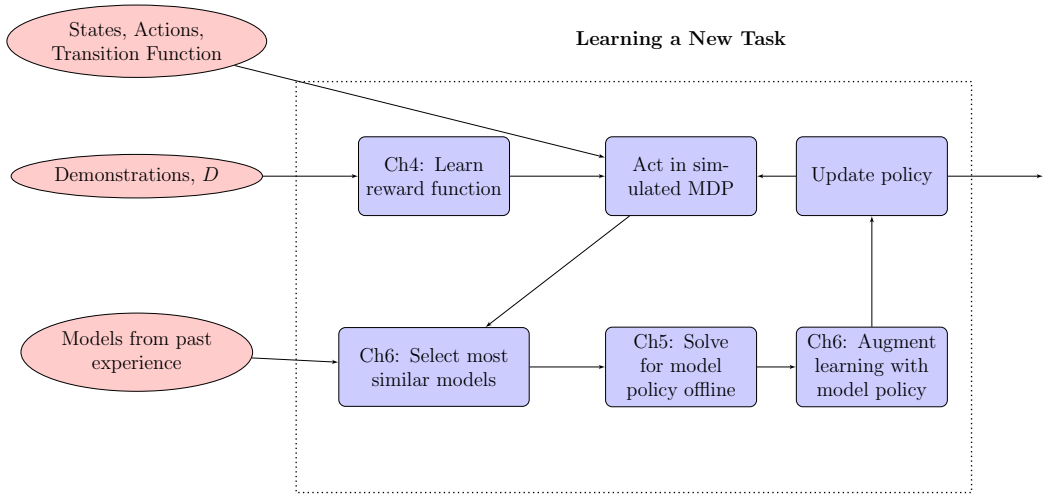

Figure 3.1: Pipeline for Multi-Task RL

3.1 Abstract Problem Space

In Chapter 2, we introduced a number of the smaller building blocks that we use. This section details how they fit into the overall problems that we address.

3.1.1 Knowledge Transfer Across Tasks

We are interested in solving a sequence of tasks generated from some underlying distribution ζ. These tasks are instantiated by a sequence of MDPs and can have varying transition or reward functions.

As an illustrative example of the generative MDP distribution ζ, imagine a simple grid world where the agent can occupy any of N^2 possible squares in some $N \times N$ region of a 2-dimensional plane. The agent has available actions $\{UP, DOWN, LEFT, RIGHT\}$.

One possibility for ζ would be with a fixed reward of 0 at all squares except one fixed goal square with a reward of 1, and fixed transition dynamics: each action moves the agent one square in the given direction, unless the agent is the edge of the grid in the chosen direction, in which case it does not move. In this case, ζ would produce the same MDP every time a new task was drawn, since the distribution is fixed in both the rewards and transitions.

A slightly more complex version of ζ could have the same fixed reward, but add noise to the transitions. ζ could choose uniformly from a noise constant $\kappa \in [0, 0.5]$. The agent would, rather than moving in a direction entirely determined by the action, instead move in a random direction with probability κ. In this case, ζ would have infinite potential outputs, each corresponding to values of κ.

Adding further complexity, ζ could still have the fixed reward, but choose source and sink squares for teleportation. When the agent's action would have placed it in the source square, it would be instead placed in the sink square. The source and sink could be chosen uniformly from the N^2 squares, so ζ would have N^4 possible outputs.

We could also combine the previous two distributions: ζ could first choose whether this environment would have transition noise or teleportation, and then from among the free parameters for the chosen structure.

We could continue on this track, twisting the possible structures of the transition function and the parameters that ζ is responsible for, and also incorporating variations on the reward function, such as the goal square changing or adding multiple goal squares. The intent of this example is to show the power of ζ, and the scope of the functions that we potentially need to account for in our policies and models.

Our goal is to learn policies to accomplish these tasks by optimizing the expected discounted cumulative reward. When the reward functions differ, the policies clearly must be different to act optimally on each task. Even when the reward functions are shared, though, a one-size-fits-all policy is untenable in a sequence of tasks with different transition functions, because different transition outcome states can have different Q-values under the Bellman equation. We have to learn different policies for each different task.

Each individual task, though, bears some similarity to those that have come before it. Humans are very good at transferring the knowledge gained in previous similar scenarios to new ones. Ideally, our RL agents would be able to determine the most relevant aspects of all previous tasks and approach each new task with a strong knowledge base. In this thesis, we make advances towards such a robust transfer system by creating algorithms to

1. Build models of tasks after they are experienced
2. Given a new task, identify the most similar models
3. Given models, find a good policy for the model-task
4. Incorporate the policy into learning for the new task

This is in addition to the problem of learning from demonstration, which we will discuss further in Section 3.1.2.

3.1.2 Learning From Demonstration

We introduced the problems of learning from demonstration, imitation learning, and inverse reinforcement learning (IRL) in Section 2.5. To summarize that section, imitation learning produces an estimate of the expert's value function or policy, while IRL produces an estimate of the expert's reward function. We chose to address learning from demonstration for this problem domain through IRL. This is not an indictment of imitation learning; it has been successfully applied to complex tasks, including the helicopter domain [14]. It also has some potential advantages over IRL: it can sometimes be applied directly to a real-world problem without an intermediate step of forward RL, and it is less likely to deviate from the expert's policy in states that the expert visited during demonstrations. This makes imitation learning more likely to produce natural-feeling trajectories.

We decided to use IRL because it seems to be a better fit for our specific problem. One strong reason for this is that the reward function produced does not necessarily depend on the transition dynamics for the specific task, while the value function or policy from imitation learning does. This would make re-using the result of imitation learning difficult across different MDPs drawn from ζ, and it would be more dependent on the specific settings present when the data was collected. Since we are trying to re-use knowledge between tasks, IRL is more attractive.

We discuss some of the challenges we faced when using IRL for this problem, primarily concerning the adaptation of CSIRL to MDPs with continuous actions, and the solutions that we found to address them, in Chapter 4.

3.2 Structure of Multi-Task Learning

Algorithm 3.1 shows the high-level flow that we follow for multi-task learning. We begin with learning from expert demonstration for each task (Step 1). This gives us the reward function for the task, which we use to create a simulated MDP (Step 4).

We maintain two arrays of models, one for transition functions and one for reward functions, that persist across all tasks (Step 2).

During a task, we take some number of steps in the MDP following policy π, interleaved with updates to π. When an update to π occurs, it starts by selecting the models that most closely match the transitions seen so far (Step 8). Next, we use the transitions stored with those models to solve an offline RL algorithm (Step 9). This gives us a policy π_M. To finish the update, we use π_M to bias policy gradient learning on π, in Step 10.

Finally, once a task is complete, we update our model arrays using the transitions that we saw during task execution (Step 12).

Algorithm 3.1 Pipeline for a Sequence of Tasks

Input: Expert demonstrations $D_{E,i}$ for each task i
Output: Policies π_i

1: $R_{E,i} \leftarrow CSIRL(D_{E,i})$ // Chapter 4. This could happen for each individual task instead.
2: Models $M \leftarrow \varnothing$
3: **for all** tasks i **do**
4: $\quad MDP \leftarrow (S, S_0, A, P, R_{E,i}, \gamma)$ // Simulated MDP
5: $\quad \pi \leftarrow INITIALIZE\text{-}POLICY$ // Begin with random or 0 policy parameter weights
6: $\quad$ **while** task is not yet finished **do**
7: $\quad\quad$ Act in MDP, following π, to collect experiences T
8: $\quad\quad M_T \leftarrow SELECT\text{-}MODELS(M, T)$ // Algorithm 6.3
9: $\quad\quad \pi_M \leftarrow CONTINUOUS\text{-}FQI(M_T, \pi.basis, \gamma)$ // Algorithm 5.4
10: $\quad\quad \pi \leftarrow GRADIENT\text{-}BIASING(T, \pi, \pi_M)$ // Algorithm 6.2
11: $\quad$ **end while**
12: $\quad M \leftarrow$ UPDATE-MODELS(M, T) // Algorithm 6.4
13: **end for**

3.3 Concrete Domains

We test our algorithms on a variety of problems to verify their effectiveness. In this section, we give the details of those problems.

3.3.1 Application to Prosthetic Arm Control

Let us consider the application of ζ on the problem of prosthetic arm motion introduced in Chapter 1. For a generalized prosthetic, a vast number of potential reward and transition functions would have to be accounted for.

A simple distribution of reward functions might be over goal points in space around the body. A more complex one might be to draw some figure on a piece of paper, selected from a library of figures.

Distributions over transition functions could involve the weight of objects held in the hand or attached to the arm, such as shirt sleeves. While most of us can move our arms without any additional effort whether we have bare arms or a heavy jacket on, the transition dynamics between those two scenarios could be different enough to warrant entirely different policies.

A diagram of the specific task that we consider, known as the Fitts' Task [15], is shown in Figure 3.2. The agent's objective is to move the pen from the region on the right to the target region on the left. Our setup is slightly different than the pictured one, because the goal regions are shown on a screen and the pen moves through free space rather than on a flat surface.

We would like to use knowledge from some trajectories to improve learning on other trajectories. Imagine that we are given the blue trajectory in Figure 3.3. How can we use that to output a controller that creates the red trajectory? The tasks are similar, but the same controller would not be able to produce both trajectories. This is what our system does.

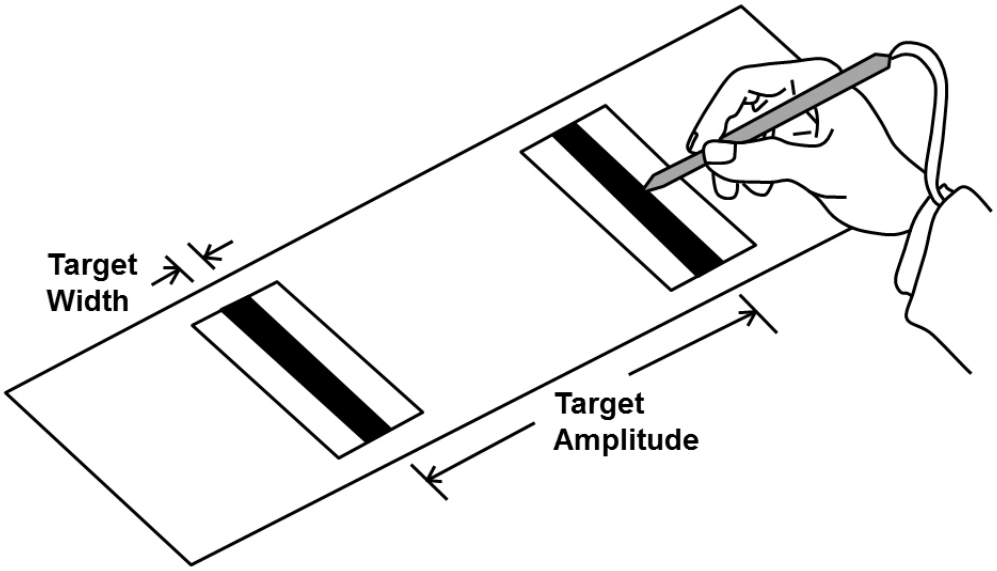

Figure 3.2: Fitts' Task [15]. The objective is to move the pen from the region on the right to the region on the left.

3.3.2 Prior Work

Our work directly builds on work by Fu [16], who set up the physical device that we used for recording. Their related contributions were variability and dynamics models for the arm and hand holding the pen. They also investigated the application of minimum-jerk models to the arm motion task that we use here, but found that they were unstable and could not form trajectories.

3.3.3 Arm Apparatus

We collected arm motion trajectories using the apparatus described in [16] and shown in Figure 3.4.

The physical interface presented to the human subject was a pen (attached to a haptic device) and a computer monitor displaying the pen's position as a cursor and the goal position as a region on the screen. As the pen moved, so did the cursor. There was no perceptual delay between the pen motion and the cursor motion. The human's task was to make the cursor reach the goal region and stop moving, and then

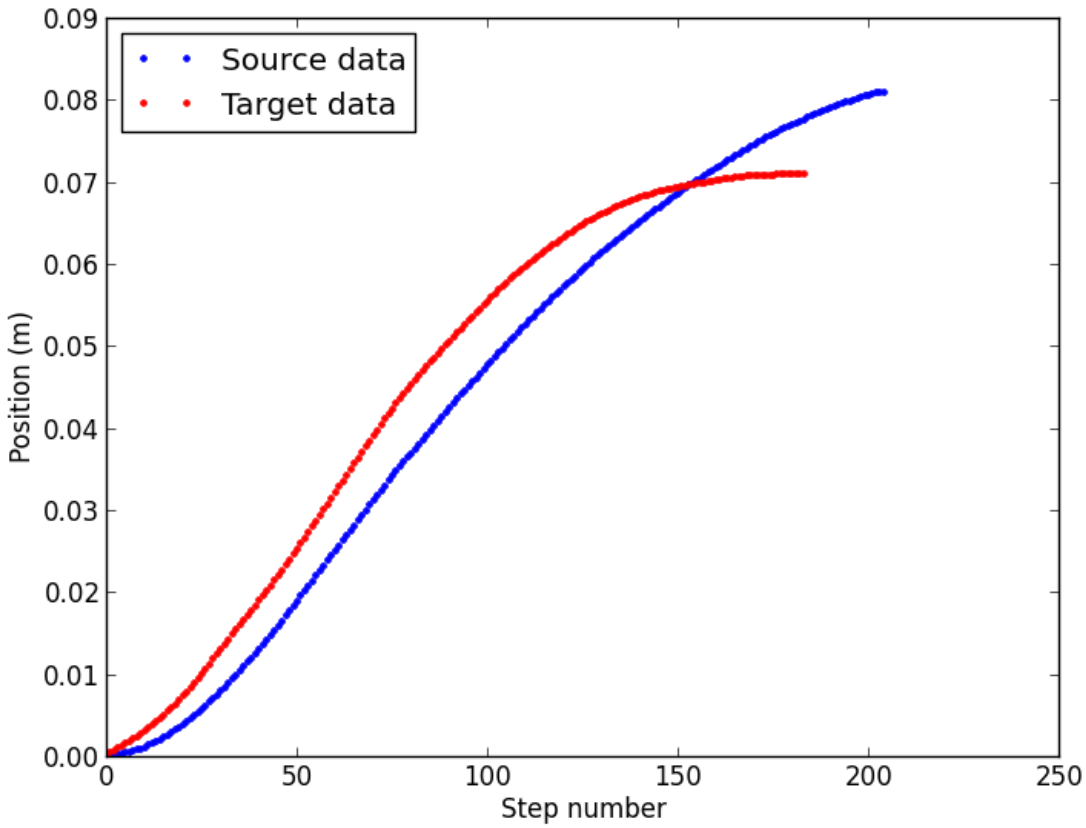

Figure 3.3: Illustration of our system goal. We would like to use knowledge from the blue trajectory to help learn to create the red trajectory.

Figure 3.4: Sensable Technologies, Inc. Phantom Premium 1.5 haptic device, taken from [16]

to press a button to signal the end of the trajectory. After the goal was reached, the human moved the cursor back to the home position at the center of the screen and pressed the button again.

The pen attaches to the device arm, which itself is attached to three angular sensors at the device base. Each of the sensors detects movement by observing a number of slots, spaced at regular angles around the range of possible motion. There are separate measurements for angular position and angular velocity. Angular velocity, which is the signal that we derive our data from, is measured by counting the amount of travel time between slots. Angular velocity readings occur at a clock rate of 80MHz.

Since the angular sensors do not directly measure the motion of the pen, but rather the motion of the device arm caused by the pen, we need to convert them to find the values for the pen motion. Those equations are given in [17]. Note that while the pen was not attached to the device in the equations in that paper, they are still applicable because we model the pen as a point mass attached to the furthest point on the device arm.

The most recent angular velocity is read from a data collection card at a rate of 1kHz, so the output recordings have 1000 data points per second. Most trajectories were finished in 2-3 seconds, leaving us with 2000-3000 data points per trajectory.

The device also has motors, capable of moving the device arm, which we used to cancel the downward gravitational force produced by the device components. This meant that the pen was the only mass that the subject had to keep aloft. We also used the motors to create "walls", forcing the pen to stay within certain boundaries in the y and z dimensions. This mostly confined movement to the x direction, which was the dimension where distance to the goal was measured.

The trajectories varied in terms of both the transition and reward function. The reward function changed by the goal moving: five different goals were used, at varying offsets from the home position. The transition function changed by introducing a re-

sistance through the haptic motor device. The resistance meant the pen moved with some velocity, the motors pushed in the opposite direction with a negative force proportional to the velocity. Trajectories were collected with a variety of proportionality constants, including zero (no resistance).

3.3.4 Data Preprocessing

While the measurements were taken in terms of angular velocity, our experiments required Euclidean positions and velocities. The angular values were converted to Euclidean coordinates via trigonometric transformations, and then the Euclidean velocities were numerically integrated to obtain positions.

We structured the arm task MDP as having state features of position and velocity, and actions of force. To obtain the force applied at each time step, we numerically differentiated the velocity to get an acceleration, and then multiplied by the mass of the pen (0.0843kg).

To smooth out a small amount of remaining recording error, we applied a moving average with a window size of 10.

While having data at a granularity of 1kHz could be helpful for other applications, it is unlikely that such fine measurements are necessary to capture the intention of human arm movement. Much of the data sampled at 1kHz is redundant. We down-sampled to 200Hz, which is more than sufficient to maintain the shape of the trajectories.

The human subject takes some time to respond at the start of each episode. This produced a period where no actions were being taken. We removed the first 30 steps from each trajectory to keep only the most relevant information. The resulting trajectories had around 200 time points each with a time step of 0.005 seconds.

3.3.5 Arm Simulator

The arm data collection apparatus is a core component of our experiments, primarily because it is a required part of the base CSIRL algorithm. Additionally, it would be difficult to achieve sufficient coverage of the state and action space through real movement since there are many possible states and actions and a human would have to perform each one. Instead of real data, we used a simulator to generate similar transitions with randomly-selected actions.

The simulator modeled the task using kinematics equations: resistance was applied by reducing the force action by an amount proportional to the velocity, $\dot{x}$:

$$f_{r,t} \leftarrow f_{applied,t} - resistance(\dot{x}_t) \tag{3.1}$$

The resisted force was then divided by the pen mass of 0.0843kg to find an acceleration, $\ddot{x}$:

$$\ddot{x}_t \leftarrow \frac{f_{r,t}}{m_{pen}} \tag{3.2}$$

The velocity, $\dot{x}$ was updated as

$$\dot{x}_{t+1} \leftarrow \dot{x}_t + \ddot{x}_t(step) \tag{3.3}$$

where *step* was the time interval that the force is applied over, in this case 0.005s. The position, x, was then

$$x_{t+1} \leftarrow \frac{\ddot{x}(step^2)}{2.0} + \dot{x}_{t+1}(step) + x_t \tag{3.4}$$

It is important to note that the simulator does not require a button to be pressed to signal the end of an episode, as is the case in the real data. It ends the episode when the position is within a small tolerance of the goal position and the velocity is within

a small tolerance of 0.

Forward RL for the arm task is run on this simulator, and any random samples are generated by it. Except where otherwise noted, all simulator trajectories ended either when the goal was reached or after 500 steps.

The goal was considered reached if the agent was within 0.0054 meters of the target, and the velocity was within $[-0.01, 0.01]\frac{m}{s}$.

3.3.6 Data Collected

There were 5 different resistances applied and 5 different goals used. Each individual combination was collected, for a total of 25 scenarios. For each scenario, we collected 10 trajectories.

The resistance constants were $\{0.0, 0.5, 1.5, 2.5, 3.0\}$, and the targets were at distances of $\{0.0528, 0.0726, 0.0792, 0.1017, 0.1081\}$ meters from the starting position.

3.3.7 Other Domains

In addition to the arm, we evaluate our algorithms on a simulated toy domain that is commonly used for RL experiments.

Cart Pole has the agent control a cart which has a pole attached to it. The objective is to keep the pole upright for as long as possible. The action available to the agent is the cart's acceleration along its one-dimensional track. The state variables that the agent has access to are the angle and angular velocity of the pole.

An episode ends when the pole falls outside the boundaries of $\left[-\frac{\pi}{15}, \frac{\pi}{15}\right]$ radians, where a vertical pole is at 0 radians. The reward is 0 for the step where the pole falls past the boundaries and 1 otherwise. The discount factor is 0.9. Episodes are terminated if they reach 200 steps.

The actions are bounded in $[-1, 1]$. In some of our experiments, we impose the constraint of discrete actions, in which case the available actions are $\{-1, 1\}$.

The initial conditions are chosen uniformly at random from $[-0.05, 0.05]$ for both the angle (θ) and angular velocity ($\dot{\theta}$).

The angle becomes:

$$\theta_{t+1} \leftarrow \theta_t + \tau\dot{\theta}_t \tag{3.5}$$

where τ is the length of the time step, 0.02 seconds. The angular velocity is updated as follows:

$$\dot{\theta}_{t+1} \leftarrow \dot{\theta}_t + \tau G \sin(\theta_t) - \cos(\theta_t)\frac{(10a + 2m_p l\dot{\theta}_t^2 \sin(\theta_t))}{l(m_c + m_p)\left(\frac{4}{3} - \frac{m_p \cos(\theta_t)^2}{m_c + m_p}\right)} \tag{3.6}$$

where G is the gravitational constant 9.8; a is the acceleration action chosen; m_c is the mass of the cart, 1.0; m_p is the mass of the pole, which varies; and l is the length of the pole, 0.5.

As mentioned in the previous paragraph, the mass of the pole varies between 0.1 and 2.0. This is how we introduce different transition functions over ζ.

3.4 Summary

In this chapter, we began with an abstract overview of the problems that we are trying to solve, including multi-task transfer learning and learning from demonstration. Then, in Algorithm 3.1, we showed how these components fit together in a pipeline for learning. Finally, we discussed the concrete domains that we will use to evaluate our algorithms.

Over the next three chapters we will present our specific algorithmic contributions and their evaluations. We begin with learning from demonstration.

Chapter 4

Learning Tree-Structured Rewards from Expert Trajectories with Continuous Actions

In the previous chapter, we discussed how we want to learn a reward function from expert demonstration using inverse reinforcement learning (IRL). Our situation is further complicated both by needing to be able to learn complex non-linear reward functions and the presence of continuous actions. In this chapter, we describe our approach and contributions for both of those issues: for the reward functions, we use model trees, which are regressors capable of learning complex decision boundaries. For the continuous actions, we alter an existing discrete-action IRL algorithm, CSIRL, to also cover the continuous case.

4.1 A Cascaded Supervised Approach to Inverse Reinforcement Learning

We will first present the standard CSIRL algorithm, and then return to our contributions later in the chapter.

4.1.1 Motivation for CSIRL

An initially tempting approach to the IRL problem, given a set of trajectories with state features and discrete actions, might be to train a predictive model with the state features as examples and the actions as labels. Then we could use the model's score function to determine how strongly it estimates that the action it chooses in a state is correct, and use that as the reward output. This would assign the highest rewards to those actions which the model was very confident the expert would take in a given state. It would assign modest rewards to actions where the choice was more ambiguous, and low rewards to actions unlikely to be chosen.

Unfortunately, the simple model-based approach has a fatal flaw. When the expert chooses actions, they have to take into account discounted future rewards as well as the immediate reward. That means the model would be unable to distinguish when the expert chose an action because it had a large immediate reward, versus when the expert chose an action because it led to high rewards in the future. We have seen this concept before in the value function. As it turns out, the $S \rightarrow A$ model can be re-purposed as part of a slightly more complex algorithm that is an effective strategy for the IRL problem.

4.1.2 Base CSIRL

The Cascaded Supervised Approach to Inverse Reinforcement Learning (CSIRL) algorithm [18], presented in an abstract form here in Algorithm 4.1, uses the $S \to A$ model as an intermediate step before computing the reward function (Step 2). Specifically, it uses the score output of this model as an approximation of $Q(s, a)$, and the output class as a decision rule (policy) $\pi^C(s)$. With these two outputs, we can rearrange Equation (2.5) to find the reward function:

$$R(s, a) = Q(s, a) - \gamma \sum_{s \in s'} P(s, a, s') Q(s', \pi^C(s')) \tag{4.1}$$

Since we are using demonstrated trajectories, we approximate the transition probabilities from the samples we have access to, and the previous equation becomes

$$R(s, a) = Q(s, a) - \gamma Q(s', \pi^C(s')) \tag{4.2}$$

This is applied to all the transitions that we have available in Step 3. Once we have a reward label for each transition, we can do a further regression operation, with state features and actions as examples, and rewards as labels (Step 5). This regressor is then output as the reward function.

One issue with this formulation is that we need to produce a reward function that has appropriate values for every state and action in the MDP. The expert transitions, though, are likely to concentrate on the states and actions with highest Q-scores, since the expert will quickly move there even from a random starting state. This means that we have poor coverage of the environment from just the expert demonstrations. This in turn has the potential to bias the reward regressor, since it will have no reason to think that un-visited states and actions are worse than those it has for examples.

The remedy proposed in [18] is to also incorporate transitions sampled with ran-

Algorithm 4.1 Base CSIRL (adapted from Klein et. al [18])

Input: Expert data-set $D_E = (s_k, a_k = \pi_E(a_k), s'_k)_k$, random data-set $D_R = (s_k, a_k = \pi_{random}(a_k), s'_k)_k$

Output: Reward function $R : S \times A \to \mathbb{R}$

1: Construct the data-set $D_C \leftarrow \{(s_i = s_k, a_i = \pi^E(s_i)) = a_k\}$
2: CONSTRUCT-SCORE-FUNCTION(D_C), obtaining decision rule π_Q and Q-score function $Q : S \times A \to \mathbb{R}$ // Choice of score function is left to the instantiation
3: Construct the data-set $D_R \leftarrow \{((s_j = s_k, a_j = a_k), \hat{r}_j)_j\}$ with $\hat{r}_j \leftarrow Q(s_j, a_j) - \gamma Q(s'_j = s'_k, \pi_C(s'_j = s'_k))$
4: HEURISTIC-EXPERT-PREFERENCE(D_R, D_E) // Optionally, add in this step to further distinguish expert actions from non-expert actions
5: $R \leftarrow$CONSTRUCT-REWARD-FUNCTION(D_R) // Learn the reward function from the training set. Again, the functional form is left to the instantiation.

dom actions. Clearly, these random samples should not be included in the Q-score classification step, or the Q-score classifier would inappropriately assign higher Q-scores to the randomly-chosen actions. Instead, they incorporate the random transitions into the inverse Bellman step. This takes full advantage of the transition information, while not deriving any action preference from them.

The theoretical results in [18] show that the expert policy, π_E, is close to optimal for the output reward function R. While this does not mean that R is guaranteed to be close to the function that the expert was intending to optimize, it does serve as a useful sanity check. Further details and proofs can be found in [18].

4.1.3 Discrete CSIRL

The implementation in [18], which we will refer to as D-CSIRL, covers the case of smooth reward functions and discrete actions.

The discrete actions are handled by the choice for CONSTRUCT-SCORE-FUNCTION (Step 2 of Algorithm 4.1), which is to train a classifier on the expert training set D_C. This classifier predicts what the action chosen by the expert would be in a given state, which is the decision rule π_Q. It also is able to give the probability that the expert would choose a given action in a state, which is the Q-score function $Q : S \times A \to \mathbb{R}$.

D-CSIRL uses a support vector machine (SVM) and a support vector regressor (SVR) for CONSTRUCT-SCORE-FUNCTION and CONSTRUCT-REWARD-FUNCTION, respectively.

There are two places in D-CSIRL where actions that the expert took in demonstration samples are given preference over other actions. The first is in the Q-score classifier: actions with higher probability of being taken by the expert in a state are given higher Q-scores.

The second place where expert actions are privileged is in Algorithm 4.2, the heuristic expert preference choice of D-CSIRL. Step 1 finds a value some fixed amount lower than the minimum existing reward over all (s, a) pairs in the expert and random data-sets. Step 2 sets that minimum value as the reward for all (s, a_n) pairs where there was at least one expert transition starting in state s and there was no expert transition where action a_n was chosen in state s.

Algorithm 4.2 Heuristic Expert Preference

Input: Bellman-based reward values D_R, Expert transitions D_E

1: Set $r_{min} \leftarrow \min_j \hat{r}_j - 1$

2: Construct the training set $D_R \leftarrow \{((s_j = s_k, a_j = a_k), \hat{r}_j)_j\} \cup \{((s_j = s_k, a), r_{min})_{j;\forall a \neq \pi_E(s_j)=a_k}\}$

4.2 Tree-Based CSIRL

As we mentioned previously, D-CSIRL handles smooth reward functions. A smooth reward function is one where an (s, a) pair having some reward value r means that nearby (s, a) pairs have a reward close to r. We want to be able to learn non-smooth reward functions, which potentially have discontinuities. We will first give reasons why this would be desirable, and then describe how we implement our solution.

4.2.1 Motivation for T-CSIRL

Some environments have reward functions that are difficult or impossible to represent by a linear function on their state features and actions.

An example of an environment for which we know a complete reward function is Mountain Car. The standard Mountain Car environment has state features of position, $p \in [-1.2, 0.6]$ and velocity, $v \in [-0.7, 0.7]$. The action is acceleration, $acc \in [-1, 1]$. The reward is 0 if $0.5 \leq p \leq 0.6$, and -1 otherwise. Clearly, there are no weights by which we could multiply a vector consisting of only the position, velocity, and acceleration to represent this reward function. Our goal is to be able to represent arbitrary reward functions.

4.2.2 T-CSIRL

One natural approach to modeling complex functions is to use trees. Trees are able to learn a wide variety of decision boundaries because of how they quickly separate the feature space as they grow deeper.

With discrete actions, decision trees suffice for the CONSTRUCT-SCORE-FUNCTION step. Standard decision trees provide label predictions and can output probabilities of labels given features, satisfying the requirements for the Q-score function.

The CONSTRUCT-REWARD-FUNCTION step requires a regressor. The approach that we take is to use regression trees, which we previously discussed in Section 2.6.1. In particular, we use model trees [7], which allow a choice of label functions to use at the leaves of the tree. The most simple of these is a constant label, but more powerful functions exist. We have chosen to use linear regressors at the leaves. This allows extremely complex functions to be learned over the output space, since there are many linear regressors, each corresponding to a small portion of the state space.

Additionally, we allow the model trees to perform ternary splits, separating the examples that are less than, equal to, and greater than the feature value. This allows the trees to immediately isolate values with anomalous rewards, and increases the complexity of function they can learn with a given depth. We could learn, for example, that there is a large spike in the reward function at a single leaf, and negligible reward elsewhere. Alternatively, we could learn a dense reward with each leaf having a different and interesting regression.

Since the function learned in the CONSTRUCT-REWARD-FUNCTION step maps (s, a) pairs to rewards, the splits can happen on either states or actions.

Recall from Algorithm 2.2 that there are three interesting functions that any tree model construction algorithm must implement: SPLIT-QUALITY, SHOULD-MAKE-LEAF, and MAKE-LEAF.

We have already covered our choice for MAKE-LEAF: we use linear regressors. When a leaf node is reached, we fit a linear regressor on the remaining examples and labels, and store it in the leaf object.

SHOULD-MAKE-LEAF is similarly fairly simple. We stop either because of node depth or because of too few samples remaining at a node. Both of these conditions are intended to help to avoid over-fitting the tree. Additionally, we stop if there is only one unique label value remaining, as remaining splits would not be able to improve the fit.

Algorithm 4.3 Should Make Leaf (Reward Trees)

Input: Node N, Samples S, Max depth D, Minimum sample number M

1: $\textit{depth-reached} \leftarrow N.depth \geq D$
2: $\textit{too-few-samples} \leftarrow |S| < M$
3: **return** *depth-reached or too-few-samples*

SPLIT-QUALITY is more complex. Since we are creating trees with linear regressors at the leaves, we make the splits based on the coefficient of determination r^2 of each split.

In general, r^2 represents the improvement in explanatory power on a data set of a linear fit on the labels over just using the average label for each sample. Predictive power is measured as the squared distance from the true label for a sample to the predicted label for that sample. Maximizing r^2 leads to the common ordinary least squares (OLS) algorithm for finding a linear regression. For a single set of features $\{x_1, ..., x_n\}$ with corresponding true labels $\{y_1, ..., y_n\}$ and predicted labels $\{p_1, ..., p_n\}$,

$$r^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - p_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2}, \tag{4.3}$$

where $\bar{y}$ is the average label over all y. In our case, we would like to measure the coefficient of determination *of a split*. Each child c of the split will have its own linear predictive fit for the $\{y_1, ..., y_{n_c}\}$ labels of the $\{x_1, .., x_{n_c}\}$ samples in its partition, as well as its own average label $\bar{y}_c$. This leads to a new formulation for r^2:

$$r^2_{split} = 1 - \frac{\sum_{c} \sum_{i=1}^{n_c} (y_i - p_i)^2}{\sum_{c} \sum_{i=1}^{n_c} (y_i - \bar{y}_c)^2} \tag{4.4}$$

To find the value that we actually choose for the split, we have to calculate r^2_{split} for each possible split value in the data set.

Note that while each child can have a different number of samples assigned to its partition, we do not need to weight each child's contribution to the sums by its number of samples. This is because the weighting is already implicitly performed by larger children having more samples contributing to the total.

4.3 Continuous Tree-Based CSIRL

The CSIRL algorithm, which we discussed in the Section 4.1.3, is theoretically well-founded and works in environments with discrete actions. However, we are primarily interested in environments with continuous actions: our motivating problem, motion of the arm, clearly has continuous actions.

It is possible to use discrete approaches to continuous data—for example, by discretizing the action space into a finite number of bins. One problem with this is the *curse of dimensionality*: if we break a single continuous variable into n bins, we now have n binary variables to deal with rather than one. This exponentially increases computational cost.

Additionally, discretization approaches encounter sampling issues. Some bins will contain many samples and some will contain few. This affects the learning problem. It also removes the ability to generalize across bin boundaries: two data points that are close together in feature space but fall into different bins will be treated entirely differently. With continuous approaches, it is possible to generalize from the strongly-sampled regions to the weakly-sampled regions. Clearly, continuous approaches at least merit investigation.

In this section, we discuss the steps that we took to convert the D-CSIRL algorithm to a version that can cover continuous actions (which we will refer to as CT-CSIRL, Algorithm 4.4).

There are two points at which the out-of-the-box D-CSIRL algorithm is not appropriate for use in an environment with continuous states and actions: in the heuristic expert preference and in creating and using the Q-score classifier.

Algorithm 4.4 CT-CSIRL

Input: Expert data-set $D_E = (s_k, a_k = \pi_E(a_k), s'_k)_k$, random data-set $D_R = (s_k, a_k = \pi_{random}(a_k), s'_k)_k$

Output: Reward function $R : S \times A \to \mathbb{R}$

1: Construct the data-set $D_C \leftarrow \{(s_i = s_k, a_i = \pi^E(s_i)) = a_k\}$

2: Train a score function-based *regressor* on D_C, obtaining decision rule π_Q and Q-score function $Q : S \times A \to [0, 1]$

3: Construct the data-set $D_R \leftarrow \{((s_j = s_k, a_j = a_k), \hat{r}_j)_j\}$ with $\hat{r}_j \leftarrow Q(s_j, a_j) - \gamma Q(s'_j = s'_k, \pi_C(s'_j = s'_k))$

4: Learn a reward function R from the training set D_R

4.3.1 Setting Minimum Rewards

Algorithm 4.2 shows one way that D-CSIRL privileges the actions taken by the expert over those that the expert never took. The intention of this operation is to shape the output reward regression function even further toward preferring those actions chosen by the expert. This is a good goal, particularly in view of the inherent IRL problem discussed in Section 2.5.1 that there are multiple reward functions compatible with the expert trajectories.

Unfortunately, Steps 1 and 2 are hard to translate to environments with continuous states and actions. Continuous states mean that transitions will rarely start from exactly the same state. Continuous actions make it impossible to create new transitions for all actions that an expert did not take in a state, since there are infinite actions. Additionally, since actions can be arbitrarily close to each other in a continuous space, this would lead to punishing actions that were effectively the same as those that the expert took.

Therefore, our approach skips the heuristic expert preference operation in CT-CSIRL.

4.3.2 Q-Score Classification

The primary incompatibility of D-CSIRL with continuous-action environments is the Q-score classification choice for CONSTRUCT-SCORE-FUNCTION, which takes a set of $(S \to A)$ pairs and produces a classifier π_C. This classifier serves two purposes in the next step of Algorithm 4.1, which is the inverse Bellman operation (Step 3). First, it gives the probability that the expert would choose action a in state s. A fundamental point in the CSIRL algorithm is that this probability is used as a proxy for the Q-function. That is, $Q(s,a) \leftarrow P_{\pi_C}(a|s)$. Most common multi-class classifiers are able to give reasonable values for these probabilities without significant modifications.

The second use of the classifier in the inverse Bellman step is to find the best Q-score that can be achieved from a given state, $Q(s,a^*)$. Conveniently, a^* is just the action with the highest probability of being chosen by the expert in state s, so it is found by simply querying the classifier for its classification output from an input of state s. This means that the best Q-score is found by taking $Q(s,\pi_C(s))$. The entire inverse Bellman equation in this context then, given a transition tuple (s,a,s'), is

$$\begin{aligned} r(s,a) &= Q(s,a) - \gamma Q(s',\pi_C(s')) \\ &= P_{\pi_C}(a|s) - \gamma P_{\pi_C}(\pi_C(s')|s') \end{aligned} \tag{4.5}$$

In the case of continuous actions, a classifier is clearly inapplicable because there are infinitely many action outputs. This suggests that we should instead use a regressor. However, while all regressors are fully capable of creating a decision rule $\pi_R : S \to A$, we also need them to be able to produce a probability $P_{\pi_R}(a|s)$. Foster and Domingos explored probability estimation techniques for classification trees in [19], but we need a regressor.

This led to us creating our own regressor for this use case, which we will describe in the next section.

4.4 Regressors

There are two steps in CT-CSIRL where we need to train a regression function on a data-set: for CONSTRUCT-SCORE-FUNCTION and for CONSTRUCT-REWARD-FUNCTION. The Q-score function maps states, parameterized by the state features, to actions. The reward function maps (s, a) pairs to real-valued rewards.

We use the reward regressor from T-CSIRL for CONSTRUCT-REWARD-FUNCTION. As we saw with the IRL reward trees, the reward regressor can learn the non-smooth reward functions in which we are interested. It is already compatible with continuous actions because the actions are features rather than labels. The splits for actions are different than in T-CSIRL because they are now a continuous feature rather than discrete, so we can do ternary splits rather than making a child for every member of the class.

We have already discussed some necessary and unusual abilities that the Q-score regressor requires in Section 4.3.2. In this section, we give the details and reasoning behind our implementation of a Q-score regressor.

4.4.1 Q-Score Regressor

The Q-score regressor has a strong influence on the reward regressor because it intermediates all the information that the reward regressor receives about the environment, except for the transition dynamics. Since our subsequent goal is to produce a tree-structured reward function, we would like to have a similar form for the Q-score function. If we used a simple regressor such as a linear least-squares approximation, we would lose the ability to discover much of the structural complexity of the reward tree.

It would be nice to be able to simply re-purpose the reward trees from Section 4.2.2, but this would be insufficient for the requirements of the Q-score function.

As introduced in Section 4.3.2, we need the ability to, given an input and an output, calculate $P(output|input)$. This led to us creating a new formulation for the leaves of the trees as well as switching splitting criteria for the construction.

The linear regressions at the leaf nodes are very useful because they give us the ability to approximate complex Q-score and reward functions. However, they are incomplete for the Q-score function because there is no way to get a probability from a standard linear regression.

Our solution here is to use linear Gaussian models at the leaves. For a given state-action (s, a_{sample}) pair, we find the action a_{reg} that a linear regression predicts given the state. Then, treating a_{reg} as the mean μ of the Gaussian distribution and the variance of the leaf as σ^2, we can find a score of a_{sample} from the increase in the cumulative density function in a small interval around it:

$$f_{CDF}(a_{sample}; \mu, \sigma^2) = CDF(a_{sample} + \delta|\mu, \sigma^2) - CDF(a_{sample} - \delta|\mu, \sigma^2) \tag{4.6}$$

To obtain the average score over the 2δ interval, we would then have to divide f_{CDF} by 2δ. However, since δ is a fixed value for a given tree, the division is not necessary for the purposes of comparing two output values. f_{CDF} is still guaranteed to give values $\tilde{P}(a_{sample}|\mu, \sigma^2) \in [0, 1]$, which works nicely for our use case.

To avoid numerical problems, we set a lower bound on the output of f_{CDF} of 10^{-20}.

A seemingly more natural approach to find such a score might be to use the probability density function of the normal distribution:

$$f_{PDF}(a_{sample}; \mu, \sigma^2) = \frac{1}{\sqrt{2\sigma^2\pi}} \mathrm{e}^{-\frac{(a_{sample}-\mu)^2}{2\sigma^2}} \tag{4.7}$$

In practice, we discovered that this could lead to a wide range of output values because at leaves with a small number of samples, the variance can give very confident

estimates of the underlying function. The definition of the PDF guarantees that the integral over the support will equal 1, but it can still give outputs much larger than 1 if the variance is small. Since we want to be able to link Q-scores together via the inverse Bellman step, having such a wide distribution because of mostly-arbitrary split differences at the leaves is not desirable behavior.

As an illustrative example, consider leaf A and leaf B for an environment with a single continuous state feature, a single continuous action, and a discount factor of $\gamma = 0.9$. The linear regression for leaf A produces an action value of 5 at state s, and the linear regression for leaf B produces an action value of 5 at state s'. However, leaf A has a standard deviation of 0.1 and leaf B has a standard deviation of 0.001. The PDF of the mean value, 5, would be 3.99 for leaf A and 398.94 for leaf B. If s' is the state produced by taking action 5 in state s and we used the PDF values for the Q-scores, then the reward for the $(s, 5)$ pair at A would be:

$$R(s,5) = Q(s,5) - \gamma Q(s',5) = 3.99 - 0.9(398.94) = -355.06 \tag{4.8}$$

The reward for $(s, 5)$ would be terrible, despite the fact that the expert clearly preferred actions close to 5 at s and similar states. Because it can give unbounded outputs, the PDF formula is very sensitive to small changes in the input data. The CDF approach does not suffer from these drawbacks.

The leaves still need to be able to find the best action given a state. For this, we have made no changes from the model trees, as the best action is still the action predicted by the fit line. MAKE-LEAF here is similar to what we described in Section 4.2.2, in that we fit a linear regressor to the examples and labels. Additionally, we then compute the covariance matrix for the distance from the true labels to the regressor predictions in order to later perform the score estimation.

SHOULD-MAKE-LEAF is unchanged: the recursive construction can be halted

by the depth of the tree, the number of remaining examples, or the lack of uniqueness of the remaining labels.

SPLIT-QUALITY is different. Since we changed the responsibilities of a leaf, we made corresponding changes to the method for selecting a split. We now have the ability to generate scores similar to probabilities:

$$Quality(split) = \sum_{\substack{node \in \\ split.children}} \sum_{s \in node.samples} \log f_{CDF}(a_s; \mu_{node}, \sigma^2_{node}) \tag{4.9}$$

f_{CDF} is given in Equation 4.6, and *node.children* is the child nodes created by the split in question. This allows us to find maximally effective splits based on our leaf models.

Note that further weighting each split quality by the number of samples per child is unnecessary, since the inner summation means that we are doing a product over all the f_{CDF}, and each f_{CDF} will be multiplied k times if there are k samples at the child. In this way, the quality measure adjusts for the sample size of each child.

4.5 Empirical Evaluation

In this section, we test that our algorithms can solve the problems that we designed them to address.

4.5.1 Hypotheses

We introduced two new algorithms in this chapter: T-CSIRL, and CT-CSIRL.

T-CSIRL is interesting because it tries to improve on an existing algorithm, D-CSIRL, by expanding the kinds of reward functions that it can learn effectively. D-CSIRL is still able to run on environments with those reward functions. We can therefore directly compare the performance of T-CSIRL and D-CSIRL on various

environments.

CT-CSIRL is interesting because it expands the kinds of environments that CSIRL-like approaches can handle at all. To our knowledge, there is no way to run D-CSIRL on an environment with continuous actions short of discretizing those actions. This means that we cannot directly compare the performance of CT-CSIRL to other algorithms.

The standard way to evaluate an IRL algorithm on an MDP with a known reward function R_M is the following:

1. Train an expert policy π_E on the MDP using some forward RL algorithm A
2. Generate expert and random samples on the MDP
3. $R_I \leftarrow IRL$(expert samples, random samples)
4. Train a new policy π_I on the MDP using A, with A receiving R_I rather than R_M
5. Compare the performance of π_I to the performance of π_E on the MDP, when both receive R_M

Since T-CSIRL can be directly compared to other algorithms, we can make stronger hypotheses about its efficacy. We hypothesize that on discrete-action environments, agents trained using a reward from T-CSIRL will achieve a higher average cumulative reward—calculated from the environment's true reward function—than agents trained using a reward from D-CSIRL.

Our hypothesis for CT-CSIRL is that an agent trained using a reward from CT-CSIRL will achieve similar average cumulative reward—calculated from the environment's true reward function—as the expert policy that generated the samples used for CT-CSIRL.

We also predict that when the agent starts from the same location as the real data trajectories, it will be able to reach the goal in a timely manner. It should also exhibit a similar trajectory shape, in terms of position and velocity, to the real data.

4.5.2 Implementation

For T-CSIRL, we used our regression trees for the reward function and decision trees from scikit-learn [20] for the Q-score function. For D-CSIRL, we used a linear regression for the reward function and a logistic regression for the Q-score function, both from scikit-learn.

4.5.3 Methodology

We used Cart Pole for the discrete-action environments for T-CSIRL evaluation, and the arm problem as a continuous-action environment for CT-CSIRL evaluation.

Cart Pole IRL was run on the three pole masses 0.1, 1.0, and 2.0. For each environment, we trained an expert policy using Deterministic Policy Gradient with a basis of 4 bins for pole angle and 20 bins for angular velocity. We ran D-CSIRL and T-CSIRL for each with 3000 expert samples and 3000 random samples.

For the arm simulator, we do not have access to the expert's complete reward function: reaching the goal region is a part of it, but it also includes information about natural motion. We instead evaluate against a sparse reward function that has a single reward upon reaching the goal state.

Arm IRL was run on each combination of resistances and targets described in 3.3.6. The input data for each combination was 3000 random sample transitions from the arm simulator and all available real arm transitions for the expert. This was 10 trajectories per combination, which totaled about 1500-2000 transitions.

The score and reward tree max depths were set to 20.

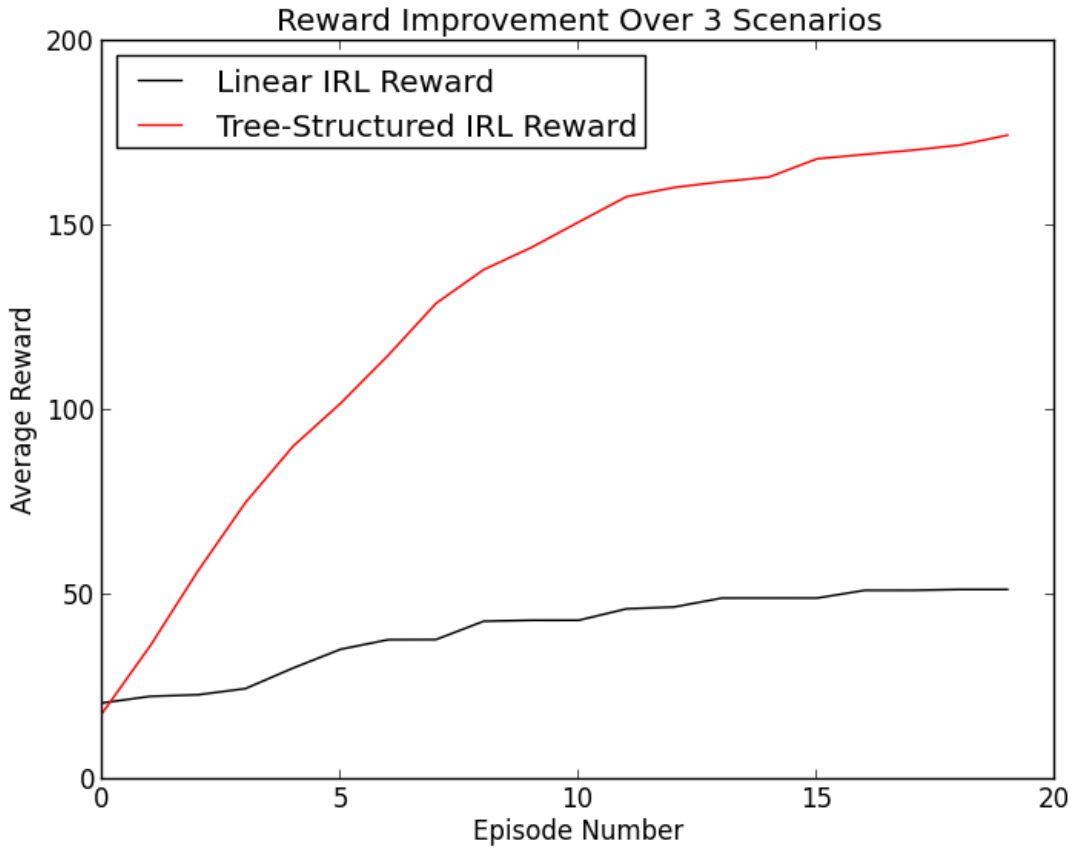

Figure 4.1: Cart Pole discrete action forward RL performance when using IRL reward function

For the arm, forward Deterministic Policy Gradient was run with the actor, critic, and baseline learning rates all being 0.001. The bases for each were tabular representations, with 12 bins for position and 12 bins for velocity, for a total of 144 bins. These parameters were obtained by manual tuning until good policies were achieved in a reasonable amount of time.

4.5.4 Results and Discussion

Figure 4.1 shows our results for T-CSIRL.

We ran 20 learners for 20 cycles each, interleaving a single learning episode with 10 evaluation episodes.

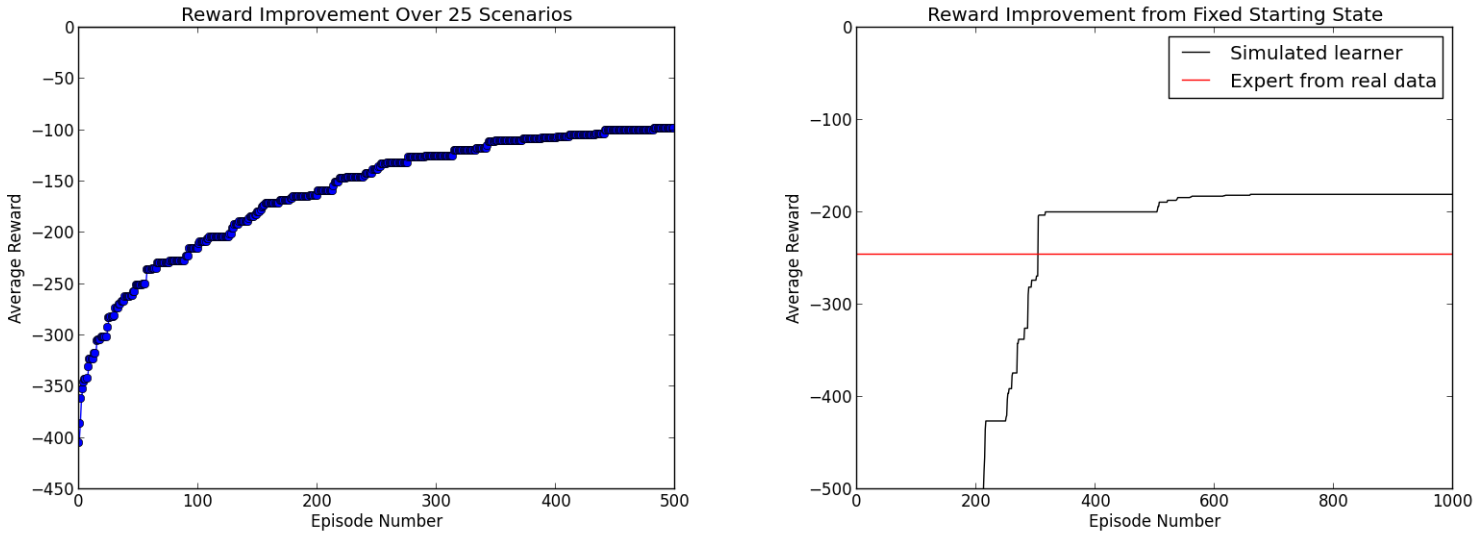

Figure 4.2: Arm improvement curve from (a) random starting configurations and (b) same starting locations as real data. (b) was only run with 3.0 resistance and the 0.1017 goal.

The results show that T-CSIRL quickly improves and approaches the maximum possible reward of 200. D-CSIRL improves slightly, but its performance is significantly worse than T-CSIRL.

This shows that our hypothesis was correct in that on environments with non-smooth reward functions, it is possible for agents trained by T-CSIRL to outperform agents trained by D-CSIRL.

Figure 4.2 shows our results for CT-CSIRL.

On the left, we ran the arm simulator using the IRL reward for training and the goal-based reward for evaluation on each combination of environment parameters. This experiment had random starting states. We ran learning for 500 cycles, with each cycle interleaving a single learning episode with 10 evaluation episodes.

On the right, we ran experiment with the evaluation starting state fixed at the same position as the real data trajectories. We ran this experiment over 2 learners for 1000 cycles each, interleaving a single learning episode with 3 evaluation episodes. This experiment used only the case of 3.0 resistance and the 0.1017 goal.

From random starting positions, the agent learns to reach the goal and stop. It also learns to do so quickly. This implies that the basic intent from the expert is indeed recovered.

From a fixed starting position, the agent also learns to reach the goal. This is a harder task than the random case, because the starting position was close to the opposite end of the state space from the goal region. The agent actually manages to reach the goal slightly faster than in the real data. This may be because the simulator does not require any sort of button press to denote the end of the episode, as was required in the real data.

To evaluate the shape of the trajectories, we trained a single learner with 0 resistance and a target of 0.0528 for 300 learning episodes. We trained and evaluated policies starting from the same state as the real data: position and velocity of 0. The policies we ran were: the policy after one learning episode, the policy the first time that the goal was reached from the zero state, and the best policy over the 300 learning episodes. We recorded the trajectories. Results are shown in Figure 4.3.

The results show that initially, the learner is unable to stop at the target position. Instead, it moves forward at a fairly constant rate until it gets close to the maximum position limit of the simulator, then slows down slightly before remaining at the maximum position for as many steps as the simulator allows.

After 139 learning episodes, the learner is able to reach the goal, but does so more slowly than the real trajectories. While the results shown in Figure 4.2 show that the learner reaches the goal on average in fewer steps than this after so many learning episodes, the current evaluation scenario (starting from the zero state) is more difficult than most starting positions. It takes more time to find a good policy.

Finally, in the best policy, the learner is able to reach the goal and stop in fewer steps than the real data. The velocity profile appears non-smooth because the actions taken are produced by the tabular basis. Since there are a limited number of bins,

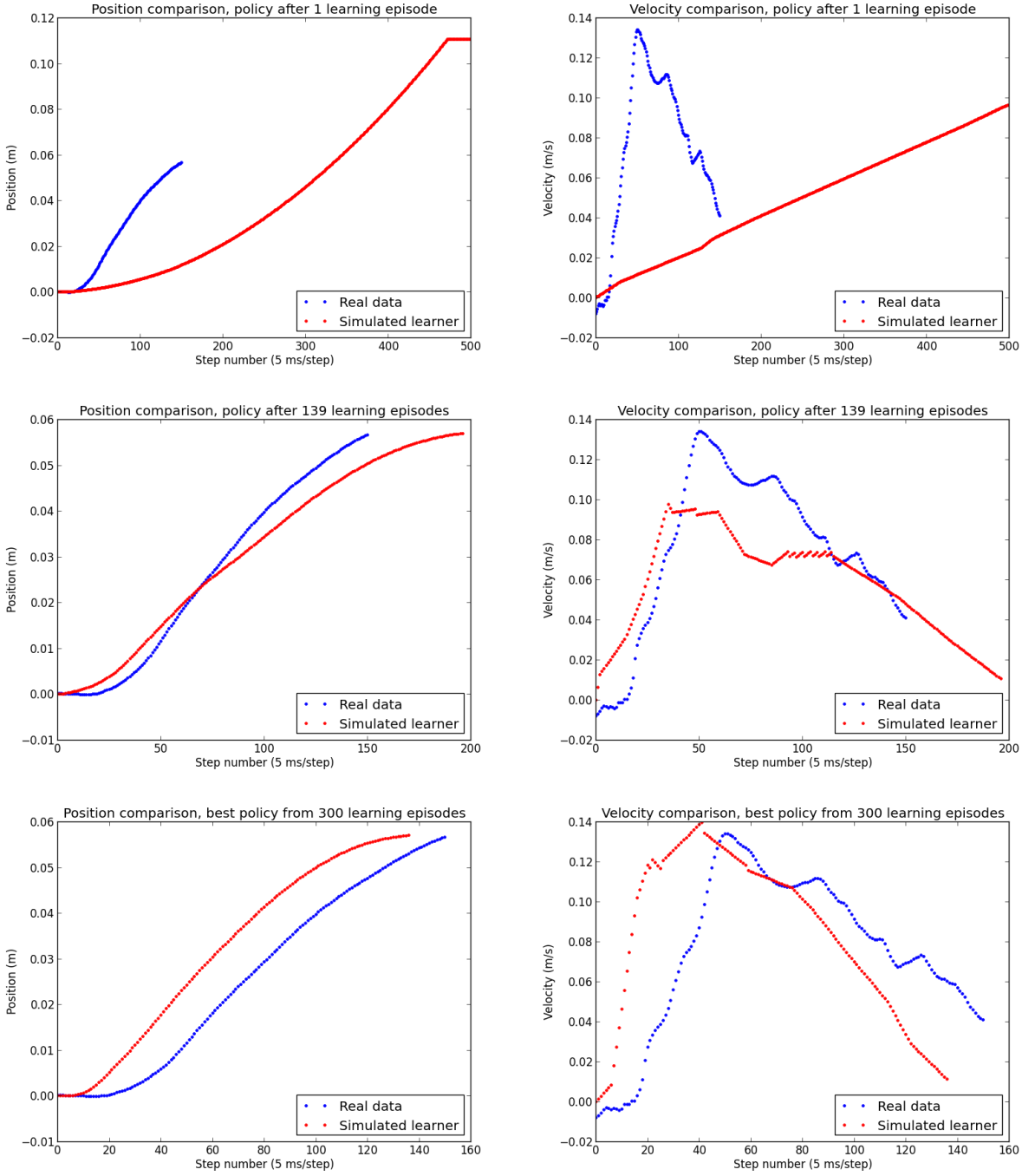

Figure 4.3: Arm trajectories, with 0 resistance and target of 0.0528.

this gives the appearance of discretized actions. However, the agent still has the full continuous range of actions available to it during learning and evaluation.

There are some interesting performance considerations for our regression tree implementations. Recall from Equation 4.4 our r^2_{split} metric for evaluating a given split, and that this metric needs to be calculated for every possible split value in the data set to find the best candidate. Thus, finding the best split has at least n^2 complexity, as for each feature and each candidate split of that feature we must do an operation over the entire data set remaining at that node (which, at the first node, is the entire overall data set).

For the linear Gaussian trees, the complexity is still n^2, but the constant and scalar costs are higher, as the covariance matrix must be computed for the full data set and two different CDF values evaluated for each sample.

It remains an interesting question whether or not there is a way to group the labels to eliminate some split values from consideration, as is possible with decision trees. We did not find such a way. To speed up computation, we implemented the relevant calculations using NumPy linear algebra operations, but tree creation is still a performance bottleneck in our experiments.

4.6 Summary

In this chapter, we covered how we find the reward function from expert demonstrations. Our contributions were a comprehensive integration of the case of continuous actions with the existing state of the art in the field, and improved ability to handle complex reward functions via tree models. In the subsequent chapters, we will talk about how we use the information available, including the reward function that we just learned, to learn to solve new tasks.

Chapter 5

Fitted Q-Iteration with Continuous Actions

In this chapter we design an algorithm to address step 9 of Algorithm 3.1. We have access to a number of transition samples for environments similar to the current one and we want to use them to create a policy. This will allow us to speed up learning on the current environment.

Assume that the samples come from an MDP that has continuous actions and a non-smooth optimal value function, as is the case in our motivating arm problem. How can we find the optimal policy for the MDP?

The standard approach to offline RL is to use Least-Squares Policy Iteration (LSPI) [5], which we discussed previously in Section 2.4.5. Unfortunately, LSPI relies on the ability to iterate through each possible action to improve its policy. This is impossible with continuous actions.

A more recent algorithm, Fitted Q-Iteration (FQI) [21], extends LSPI to use non-linear regressors to model the value function. However, the standard FQI algorithm also relies on discrete actions, and to our knowledge there is no existing FQI extension that would be applicable to our situation.

Our contribution in this chapter is a continuous-action version of FQI that can learn non-linear value functions through the use of model trees. Before we present our algorithm, though, it is important to first understand the standard FQI algorithm, which we discuss in Section 5.1.

5.1 Fitted Q-Iteration

In Section 2.4.5, we discussed the LSPI algorithm for finding optimal policy parameters during offline RL. LSPI is useful, but its representation of the Q-function is by definition limited to a least-squares regressor parameterized by the basis function. Since we want to be able to learn good policies in environments with complex reward functions, we need regressors capable of fitting more complex functions on the state variables.

Ernst et al.'s Fitted Q-Iteration (FQI) [21] algorithm allows us to use such regressors. The standard FQI implementation (Algorithm 5.1) is very similar to LSPI. It takes a set of (s, a, s', r) samples and returns an optimal policy for the environment.

In Step 1, the Q-values for all states are initialized to 0, since no information about them is known yet. The main loop, starting with Step 2, then runs for a fixed number of iterations or until some convergence test is met. On each loop iteration, the Q-values of the prior states are updated via the Bellman equation (Step 3). Next, in Step 4, some regression function is fitted from the states and actions to the new Q-values. Finally, in Step 5, the Q-values of the post-states are updated by iterating over each action and choosing the Q-value of the maximizing action. Once the loop ends, the policy is found by a similar procedure, choosing the action that produces the best Q-value for each state.

The FQI algorithm in Algorithm 5.1 is incompatible with continuous actions for one primary reason: it is impossible to iterate over all values in a continuous interval.

Algorithm 5.1 FQI

Input: Samples $(s, a, s', r)_{1..n}$, Number of iterations j, Discount factor γ
Output: Actions $a^{\pi}_{1..n}$

1: $Q^{\pi}_0(s')_{1..n} \leftarrow 0$
2: **for all** $k \in 1..j$ **do**
3: $\quad Q_k(s,a)_{1..n} \leftarrow r_{1..n} + \gamma Q^{\pi}_{k-1}(s')$
4: $\quad REG_k \leftarrow CREATE\text{-}REGRESSOR((s,a)_{1..n}, Q_k(s,a)_{1..n})$
5: $\quad Q^{\pi}_k(s')_{1..n} \leftarrow \max_{a' \in A} QUERY\text{-}REGRESSOR_k(s',a')_{1..n}$
6: **end for**
7: $a^{\pi}_{1..n} \leftarrow \arg\max_{a \in A} QUERY\text{-}REGRESSOR_j(s,a)_{1..n}$
8: **return** $a^{\pi}_{1..n}$

Therefore, finding the best Q-value and the best action in a continuous action space necessarily requires a different or altered approach.

The implementations presented in [21] only allow for MDPs with discrete actions. Versions of FQI capable of handling continuous-action MDPs do exist, such as Antos et al.'s work in [22], which relies on a search over a candidate policy set. However, that algorithm makes restrictive assumptions about the state variables, and their theoretical results are based on linear regressors. Additionally Busoniu et al. [23] presented a continuous-action version of LSPI. This is different from what we are looking for because we use tree representations. Since no existing implementations fit our needs, we designed our own modifications to the FQI algorithm that allow it to handle continuous actions. We cover them in the remainder of this chapter.

5.2 Fitted Q-Iteration with Continuous Actions

In this section, we present the alterations that we made to create a continuous version of FQI, as well as a novel addition to the FQI Bellman backup operation that serves as both a performance improvement and a stopping criterion. The full Continuous FQI algorithm is detailed in Algorithm 5.4.

5.2.1 Maximizing the Q-Function

There are two points in the standard FQI algorithm where application to continuous actions is non-trivial. First, we need $\max_{a \in A} Q(s, a)$ to use in the Bellman update expression. Second, we need $\arg\max_{a \in A} Q(s, a)$ to find $\pi_{N+1}(s)$.

With discrete actions, the values above are easy to find because for a given state s, we can calculate $Q(s, a)$ for each $a \in A$ and take the maximizing a and $Q(s, a)$. With continuous actions, such an exhaustive search is not possible.

The solution we use was first suggested by Ernst et. al [21], but to our knowledge it had never been implemented before now.

The solution brings us once again to the idea of regression trees, with features of state variables and action components and labels of Q-values. However, instead of using model trees with linear regressions at their leaves as we discussed in Section 4.2.2, we use single scalar values for the label of each leaf. The trees use ternary splits as described in Section 2.6.2, so the value tests can be $<, >, =, \leq$, and $\geq$.

The idea proposed by Ernst et. al is to note that given s, such a tree gives us a piece-wise functional description of $Q(s, a)$ over the values of a. Furthermore, the number of distinct regions of the support of that function is finite, limited by the depth of the tree and the degree of fan-out. Given the root node of such a tree and a state s, we can find $\max_{a \in A} Q(s, a)$ by following the recursive procedure on the nodes of the tree given in Algorithm 5.2.

The base case, in Step 2, is simple: at a leaf, the best Q-value that can be achieved is the Q-value label of that leaf.

Otherwise, there are two cases. If the current node splits on a state feature (Step 5), then there is only one child that we can reach because the input state is fixed.

If, instead, the current node splits on an action (Step 12), then we make a recursive

call to each of the children, since any action available as a child could turn out to be the action with the highest Q-value.

Algorithm 5.2 Q-MAX

Input: Root node r, State s
Output: $\max_{a \in A} Q(s, a)$

```
1: if r.leaf then
2:    return r.label
3: end if
4: FEATURE-TYPE ← TYPE(r.feature) // Does this node split on states, or actions?
5: if FEATURE-TYPE is STATE-VARIABLE then
6:    for all child ∈ r.children do
7:       // Iterate over the children, looking for the one that the state belongs to
8:       if child.membershiptest(s) then
9:          return Q-MAX(child, s)
10:      end if
11:   end for
12: else
13:   // The current node splits on actions
14:   return max_{child∈r.children} Q-MAX(child, s)
15: end if
```

Finding $\arg\max_{a \in A} Q(s, a)$ requires only a small modification to Algorithm 5.2. As we move down the tree, we track the action constraints that have appeared so far. When an action split occurs, we copy the existing constraints for each branch and modify them to take into account the new split. At a leaf, rather than just returning the leaf's Q-value label, we return both the Q-value and the action constraints. Finally, given the action constraints returned by the root node, we create an action that conforms to those constraints by either sampling uniformly at random from the given remaining range for each action component or by selecting the value at the middle of the range.

5.2.2 Action-Based Over-fitting

The tree-structured continuous FQI algorithm in the previous sections has a problem: over-fitting can occur during tree construction. This in turn leads to the Q-MAX

function (Algorithm 5.2) allowing high Q-values to be assigned to states that should in fact have low Q-values.

To illustrate the problem, consider a toy MDP with two state variables, position and velocity, and one action, acceleration. The position, (x) is bounded between $-1.0m$ and $1.0m$, the velocity ($\dot{x}$) between $-0.1\frac{m}{s}$ and $0.1\frac{m}{s}$, the acceleration ($\ddot{x}$) between $-0.01\frac{m}{s^2}$ and $0.01\frac{m}{s^2}$, and a time step of one second. The reward function gives 1 if $-0.9 < x < 0.9$ and 0 otherwise. The episode ends if the 0 reward is received. The discount factor is $\gamma = 0.9$.

After several iterations of tree creation and Q-value updates, we arrive at an internal node with the following conditions above it in the tree: $x < -0.2, \dot{x} < -0.04, x < -0.5$. The data points remaining at the node are:

Row	x	$\dot{x}$	$\ddot{x}$	Q
0	-0.85	-0.06	-0.001	0.03
1	-0.52	-0.098	0.002	3.1
2	-0.82	-0.042	-0.0075	2.54
3	-0.65	-0.091	-0.0032	2.54
4	-0.55	-0.053	0.0086	7.12
5	-0.59	-0.045	0.0095	7.12

There are clearly two outliers among the Q-value labels: rows 4 and 5. They have a combination of relatively high position and relatively high velocity that allows them to move back towards the center of the state space rather than having the episode inevitably end. These points are not immediately separable by the state features, x and $\dot{x}$, because rows 4 and 5 do not have the extreme values for either feature. Instead, the best split in this situation uses $\ddot{x}$ to separate the last two rows from the rest of the points, creating two child nodes: one with $\ddot{x} < 0.0086$ and further splits below it, and a leaf with $\ddot{x} \geq 0.0086$ and $Q = 7.12$.

While this is the best immediate split, it creates a problem. On the next iteration of the algorithm, any (s, a) pairs satisfying the conditions above this node ($x < -0.2, \dot{x} < -0.04, x < -0.5$) will reach this node. Then they will be able to choose an

action $\ddot{x} \geq 0.0086$ and achieve a good Q-value of 7.12. However, many of those (s, a) pairs should in fact have low Q-values, as the other points that created the split do. For example, a state with $x = -0.88$ and $\dot{x} = -0.07$ will be able to take an action $\ddot{x} \geq 0.0086$ and receive a Q-value of 7.12, when it should have a Q-value of 0.

The effect of this is that the good Q-values become spread farther across the state space than they should. Once this situation is reached, it is difficult to recover because we are performing offline learning, not online, and new samples are not introduced to correct the problem. The best actions may not be given preference over other actions, and a good policy may not be achieved.

Our solution to this issue is to initialize the Q-tree with the Q-values of the policy that generated the samples. We do this by first performing a number of iterations with actions excluded so that the Q-function is only parameterized by the states. This means that a maximization cannot be performed over the actions, since all actions will lead to the same leaf.

Algorithm 5.3 Initialize Q-Function of Sampling Policy

Input: Samples $(s, a, s', r)_{1..n}$, Discount factor γ, Number of iterations m
Output: Regression Q-tree $S \rightarrow \mathbb{R}$

1: $Q_0(s') \leftarrow 0_{1..n}$
2: **for** $k \in 1..m$ **do**
3: $\quad Q_k(s)_{1..n} \leftarrow r_{1..n} + \gamma Q_{k-1}(s')$
4: $\quad T_k \leftarrow BUILD\text{-}TREE(s, Q_k$
5: $\quad Q_k(s') \leftarrow QUERY\text{-}TREE(T_k.root, s')$
6: **end for**
7: **return** $Q_m(s')$

The idea is that the Q-function of the generative policy will be much closer to the Q-function of the optimal policy than an initial Q-function of 0 for all samples. Once we have the Q-function of the generative policy, we perform the full Continuous FQI algorithm as described previously, incorporating actions. The advantage of this approach is that fewer iterations of the full algorithm are required, lessening both the chances for action-based over-fitting to occur and the impact if it does occur.

5.2.3 Effective Sampling

Training on samples generated by a purely random policy can reduce the chances of convergence on some environments.

One problem with random sampling in the 'connectedness' of the samples: if the optimal policy must follow some path to reach a goal state and a step along that path is not present in the training samples, then the optimal policy will be impossible to find.

For example, consider a simple environment with states $\{1, 2, 3\}$ and a terminal goal reward at state 3. The agent can move to state 2 from state 1, and to states 1 or 3 from state 2. Suppose that the samples provided contain no transitions from state 2 to 3. State 3 would have a good Q-value, since the samples that start there immediately receive the goal reward and terminate. However, states 1 and 2 would never make a Bellman update with the state 3 reward, so their Q-values would remain low. An offline algorithm would not discover the optimal Q-function.

Under a random sampling policy, this is most likely to occur in 'bottleneck' areas of the state space where it is difficult to pass from one region to another by chance alone. If these key transitions are not present in the training samples, then the trees will keep the regions separated and the necessary Bellman updates will not occur. Bottlenecks are a problem even in discrete state and action spaces, and with continuous states and actions the issue is compounded further.

While this could potentially be mitigated by using a very large number of training samples, training the Q-trees was already a significant performance issue in our experiments and it would have become even worse.

Our approach was to add in samples generated by an expert policy. The samples could be reused from an expert demonstration such as the one in Chapter 4, or from a previous task where we learned a good policy. The samples effectively guide

the exploration of the algorithm, improving the chances that the most important transitions are available to it.

Algorithm 5.4 Base Continuous FQI

Input: Samples $(s, a, s', r)_{1..n}$, Target basis $\phi(s)$, Discount factor γ, Number of iterations $N_{terminate}$

Output: Actions $a^{\pi}_{1..n}$

1: $Q_0(s,a)_{1..n} \leftarrow INITIALIZE\text{-}Q\text{-}FUNCTION((s,a,s',r)_{1..n}, \gamma)$ // Algorithm 5.3
2: **for** $k \in 0..N_{terminate}$ **do**
3: $\quad T_k \leftarrow BUILD\text{-}TREE(s, a, Q_{k-1})$
4: $\quad Q^{\theta}_k(s')_{1..n} \leftarrow Q\text{-}MAX(T_k.root, s')$
5: $\quad Q_k(s,a)_{1..n} \leftarrow r_{1..n} + Q^{\theta}_{k-1}(s'_{1..n})$
6: **end for**
7: $a^{\pi}_{1..n} \leftarrow \arg\max_{a \in A} QUERY\text{-}TREE(s,a)_{1..n}$ // Find the actions for each state
8: **return** $a^{\pi}_{1..n}$

5.3 Experiments

The Continuous FQI algorithm creates a tree-structured Q-function. We can use that Q-function as a tree-structured policy π_{tree} through the $Q\text{-}MAX$ algorithm presented in this chapter.

We are also able to generate a number of $(s, \pi_{tree}(s))$ pairs. This lets us run a regression with examples of some other basis $\phi(s)$ and labels of $\pi_{tree}(s)$). The output of that regression is a policy of the form $\pi_{\phi}(s)$. In Chapter 6, we will see that we need to perform this operation to obtain a policy in the same tabular basis as the policy we use for forward RL.

In this section, we evaluate both π_{tree} and $\pi_{tabular}$.

5.3.1 Hypotheses

We expect π_{tree} to show improvement over our baseline, which is the random policy. This improvement should increase with more iterations of the Continuous FQI algo-

rithm, and eventually should approach the performance of the forward RL results in Section 4.5.4.

$\pi_{tabular}$ should also show improvement over the random policy.

A direct comparison between π_{tree} and $\pi_{tabular}$ is an interesting question. π_{tree} benefits from having a much more fine-grained representation of the state space when the tree depth is large. $\pi_{tabular}$, however, could potentially eliminate over-fitting in the π_{tree} Q-value regressor. Additionally, while the output policy of $\pi_{tabular}$ is compact, it still benefits from the richness of the tree-structured Q-functions used at intermediate stages of the Continuous FQI algorithm. We expect π_{tree} and $\pi_{tabular}$ to be close in performance.

5.3.2 Methodology

The basic experimental structure for evaluating an offline RL algorithm is straightforward: we give the algorithm some number of samples to find a policy and then evaluate that policy on the target environment.

For the arm simulator, we used the IRL reward from Section 4.5 to generate the training samples and evaluated on the simple goal-based reward.

Continuous FQI was run with max tree depth of 30. This was intended to be effectively unlimited depth, since we observed that the trees only ever reached depth of approximately 22.

Random samples were generated by running uniform random policies over the environment with uniform random starting states and episodes of maximum 10 steps.

Expert samples were generated by using the best policies from the experiments in Section 4.5 for each environment configuration.

The tabular basis for the arm simulator, similarly to Section 4.5.3, had 12 bins for position and 12 bins for velocity, for a total of 144 bins.

5.3.3 Results and Discussion

The parameters that varied for each arm experiment were the environment parameters (resistance res and goal location $goal$); the number of random samples, s_r; number of expert samples, s_e; number of initialization iterations for Algorithm 5.3, N_{init}; and number of full Continuous FQI iterations for Algorithm 5.4, N_{FQI}. Results are given in Table 5.1.

Row	res	$goal$	s_r	s_e	N_{init}	N_{FQI}	π_{tree}	$\pi_{tabular}$	π_{random}
0	0.5	0.1081	10000	5000	100	5	-137.1	-495.0	-394.9
1	1.5	0.1017	5000	5000	50	20	-430.9	-418.2	-486.0
2	3.0	0.0726	5000	5000	50	20	-306.3	-267.8	-493.4

Table 5.1: Arm Simulator Continuous FQI results. π_{tree} is the policy from the Q-tree. $\pi_{tabular}$ is the policy obtained by linear regression from π_{tree} to a tabular basis.

In Row 0, π_{tree} significantly outperformed the random policy and came close to the best results from Section 4.5.4. This matched our hypotheses. $\pi_{tabular}$ did worse. This did not match our hypotheses. The discrepancy could be due to an inopportune bin boundary, where an averaging of high actions and low actions causes the agent to stop just short of the goal region.

In Rows 1 and 2, π_{tree} and $\pi_{tabular}$ both outperformed the random policy. This matched our hypotheses. They did not perform as well as the best policies from Section 4.5.4. Since π_{tree} did much better with more random samples and more initialization iterations in Row 0, those are the best candidates to explain the discrepancy and an opportunity for further exploration.

$\pi_{tabular}$ did perform better than π_{tree} in Rows 1 and 2, but they were fairly close as we hypothesized. These results suggest that it is possible for the policy to improve from being forced into a more compact representation.

In Table 5.2, we show the results for running Continuous FQI on Cart Pole with

various pole masses. The tabular basis had 4 bins for the pole angle and 20 bins for the angular velocity.

Row	$mass$	s_r	s_e	N_{init}	N_{FQI}	π_{tree}	$\pi_{tabular}$	π_{random}
0	0.1	2000	1000	20	5	180.046	200.0	26.238
1	1.0	2000	1000	20	5	61.374	183.754	26.608
2	2.0	2000	1000	20	5	137.334	187.736	26.524

Table 5.2: Cart Pole Continuous FQI results. π_{tree} is the policy from the Q-tree. $\pi_{tabular}$ is the policy obtained by linear regression from π_{tree} to a tabular basis. Maximum possible reward is 200.

The results show that both π_{tree} and $\pi_{tabular}$ outperform the random policy by a large margin on all scenarios. $\pi_{tabular}$ does better than π_{tree} in all scenarios. This supports the notion that the results can potentially be improved by forcing the policy into a more compact representation.

5.4 Summary

This chapter covered how we use previous knowledge to find good policies for an environment. Our contributions included an extension of existing discrete offline RL algorithms to the continuous case. In the next chapter we will discuss how we match the current environment to our prior knowledge and how we use the policy output of the offline RL algorithm to improve learning on the current environment.

Chapter 6

Model Reuse for MTRL

Let us return to the motivating problem of learning natural arm motion. We would like to observe human demonstrations and be able to reproduce their intent, or reward function (covered in Chapter 4). For example, this could correspond to the task of moving the arm between points A and B. With the reward function in hand, we can use an online RL algorithm to find a policy $\pi_{A,B}$ that moves the arm according to the human intent.

When we want to perform a new task (say, moving between points C and D), we could just repeat the same online RL procedure and find a new policy $\pi_{C,D}$. $\pi_{A,B}$ and $\pi_{C,D}$ would be markedly different, since they have different goals.

However, this would be inefficient. When a human learns to move their limbs, they do not have to learn a new policy from scratch for each new pair of points they want to traverse. Rather, they transfer knowledge from the tasks they have performed previously to the new one.

This applies to the case of the prosthetic arm. Imagine that a person has figured out how to use their prosthetic to move between points C and D, holding a pen. What happens when the weight of the pen changes? Clearly, they should not have to start anew. They should be able to use what they had learned previously to guide

improvement on the new, similar task.

In this chapter, we present a strategy for performing knowledge transfer (i.e., multi-task reinforcement learning) via model reuse. We construct models of the environments we have seen previously and then re-use them in future environments to speed up learning. Between tasks, we update two different arrays of models: one for the transition functions we have seen previously and one for the reward functions.

Algorithm 6.1 details our entire approach. Imagine that we come to a new environment and we can immediately identify an equivalent environment that we saw in the past. Since we acted on the equivalent environment previously, we had found a good policy π_M for it. In that case, we could just use π_M as the final output policy for the new environment. We discussed prior work on policy reuse in Section 2.7.3. Unfortunately, policy reuse is impractical for the problems we address because of two unrealistic conditions.

The first issue is that we may not have seen an identical environment previously. Even if the most similar previous environment is very close to the current one, with only slight changes in the transition or reward functions, π_M is unlikely to be optimal. This means that to find an optimal policy for the new environment, we will need to perform some amount of online learning to make specific adaptations. We have found a way to do the online learning, while still incorporating the knowledge that we get from π_M. To our knowledge, there are no existing online RL algorithms that can make use of models in the manner that we have constructed them. We will describe our procedure, which we refer to as gradient biasing, in Section 6.2.

The second issue is that we cannot immediately identify the best environment that we have previously seen. This is true for humans as well: we do not have much information about a task before we have tried it. We have to collect some experience in the new environment before we can make any sort of comparisons. As we gain more experience, our accuracy and confidence about the best previous environment

increases.

One question that naturally arises is how we select the best model given the arbitrarily large possible range of different underlying transition and reward functions for MDPs. It would be insufficient to try to identify individual parameters that vary between tasks. Our approach is to use tree models to perform an implicit discretization on the MDP functions. This leads to the problem of matching, which we will explore further in Section 6.3.1. Additionally, we need a long-term management strategy for the different models that we create. This is described in Section 6.3.2.

The output of the matching procedure is an environment with associated transition samples. The input to the gradient biasing procedure is a policy. That gap is bridged by an offline learning algorithm finding the best policy for the matched environment. We presented such an algorithm in Chapter 5.

As we gain more experience in the new environment, the output of the matching procedure improves and π_M becomes better suited to the new environment. Therefore, we run the entire procedure (matching, FQI, and gradient biasing) at multiple steps. We expand our notation to $\pi_{M,k}$ to denote the policy found at the k-th step.

6.1 Algorithmic Pipeline for a New Environment

Algorithm 6.1 details how we find the best policy parameters for the model worlds. For most environment execution steps k, we simply perform a standard policy gradient update (Step 13).

When we decide to do a full update, we begin by using the models from previous knowledge to create transition and reward samples that are similar to the current environment (Step 8). Next, we use those transitions to find a good policy for those samples (Step 9) and transform that policy to be compatible with our current policy (Step 10).

After finding the best compatible policy parameters, we use them to bias our current policy (Step 11).

Algorithm 6.1 Model-Augmented Learning for New Environment

Input: Models M, Basis function ϕ, Learning rate α
Output: Updated policy parameters, θ_{k+1}

1: $T \leftarrow \varnothing$ // Set of recorded transitions for the environment
2: $\pi_0 \leftarrow INIT\text{-}POLICY(\phi)$ // Initialize the policy somehow
3: $\lambda_0 \leftarrow INIT\text{-}BIASING\text{-}RATE$ // Section 6.2.4
4: **for** Step $k \in$ Environment Execution **do**
5: $\quad T \leftarrow T \cup \{transition_k\}$
6: $\quad \lambda_k \leftarrow UPDATE\text{-}BIASING\text{-}RATE(\lambda_{k-1})$ // Section 6.2.4
7: $\quad$ **if** $SHOULD\text{-}RUN\text{-}PROCEDURE(k)$ **then**
8: $\quad\quad M_T \leftarrow SELECT\text{-}MODEL(T, M)$ // Algorithm 6.3
9: $\quad\quad s_k, a_{\pi,k} \leftarrow CONTINUOUS\text{-}FQI(T, M_T)$ // Algorithm 5.4
10: $\quad\quad \theta^*_{M,k} \leftarrow LINEARIZE\text{-}POLICY(\phi, s_k, a_{\pi,k})$ // Algorithm 6.5
11: $\quad\quad \pi_{k+1} \leftarrow GRADIENT\text{-}BIASING(T, \pi_k, \theta^*_{M,k}, \lambda_k, \alpha)$ // Algorithm 6.2
12: $\quad$ **else**
13: $\quad\quad \pi_{k+1} \leftarrow VANILLA\text{-}POLICY\text{-}GRADIENT(T, \pi_k, \alpha)$
14: $\quad$ **end if**
15: **end for**

6.2 A Model Augmentation Step for General Policy Gradient Algorithms

Imagine that we have selected models similar to the current environment, and we have used FQI to find a policy $\pi_{M,k}$ that achieves a good expected return on the environment created by those models. How can leverage $\pi_{M,k}$ to help find a good policy for the current environment?

6.2.1 Motivation for Policy Gradient Augmentation

One tempting approach might be to just use $\pi_{M,k}$ as the final output policy for the new environment. If this was possible, we would not even need to obtain any more samples from the current environment. Unfortunately, this is not a good idea.

Recall that the solved policy was created from samples of previous transition and reward functions. Those previous functions were selected from the model lists by the matching procedure in Algorithm 6.3. As we observe more transition and reward samples from the current environment, the matching procedure may decide that a different model is better suited to the current environment than one that it previously chose. When a low number of samples have been recorded, the matching procedure is particularly inaccurate and volatile. If we stopped collecting samples after we had created a solved policy, then that policy would potentially be from a world that was not as close as possible to the current environment.

Clearly, we need to at least collect samples until we are confident about which models are the best match. Why, though, do we need to do further learning after the matching procedure has converged? The answer is that even the best models may not exactly match the current environment; they are merely similar. The policy obtained by the offline learning algorithm, then, is not necessarily optimal for the current environment, and we should do further online learning to adapt the current policy to the current environment.

6.2.2 Modified Target Function for Proximal Optimization

Policy gradient algorithms, which we discussed in Section 2.4.3, optimize their target function, the cumulative discounted expected return $J(\theta)$, by moving the policy in the direction of the gradient $\nabla_\theta J(\theta)$.

Most optimization algorithms aim to find the global maximum of some function over its support. Proximal optimization algorithms [24] find the maximal point of a function within the vicinity of some fixed point. That is, the best solution that is close to some fixed other solution.

Our contribution here is an application of proximal optimization to policy gradients. By modifying $J(\theta)$, we are able to find the best policy within the vicinity of the

solved policy. $J(\theta)$ becomes:

$$J(\theta)^+|_{\theta=\theta_k} = J(\theta)|_{\theta=\theta_k} - \frac{\lambda_k}{2}(\theta^*_{M,k} - \theta_k)^2 \tag{6.1}$$

$J(\theta)$ is the standard policy gradient target. λ_k is the biasing rate, controlling how much the model input bias is weighted compared to the input from $J(\theta)$, and it can be tuned and enhanced in various ways (see Section 6.2.4). The real informational content is in $(\theta^*_{M,k} - \theta_k)^2$. This represents the difference between the best policy parameters found from the input models, $\theta^*_{M,k}$, and the current policy, θ_k. By including this in the target function, we incentivize the update procedure to decrease the distance between θ_k and $\theta^*_{M,k}$.

Interestingly, this modified target includes the current policy parameters, and means that by following $\nabla_\theta J(\theta)^+$, the expected discounted cumulative return of π will converge to a value different from $J(\theta)$, influenced by the chosen model. However, by decaying λ_k as we will discuss in Section 6.2.4, we can make that difference shrink as more experiences are collected from the environment.

Note that the solved policy parameters need to only be a small amount better than the random initial parameters to show benefit, since better parameters will allow the agent to immediately achieve a better reward.

6.2.3 Gradient Biasing

To incorporate the modified target $J(\theta)^+$ in the policy gradient algorithm, we need to find the modified gradient:

$$\nabla_\theta J(\theta)^+|_{\theta=\theta_k} = \nabla_\theta J(\theta)|_{\theta=\theta_k} + \lambda_k(\theta^*_{M,k} - \theta_k) \tag{6.2}$$

The full modified policy gradient algorithm is given in Algorithm 6.2. Step 1 finds the gradient update that the base policy gradient algorithm would apply given the

experiences. $VANILLA\text{-}POLICY\text{-}GRADIENT(\theta_k, T)$ is a stand-in for the base algorithm's method of computing the gradient. We will give our implementation-specific equations for this in Section 6.6.1. Step 2 is where the biasing takes place. We augment the standard gradient value to incorporate the model information. Finally in Step 3 we calculate new weights by adding the augmented gradient, scaled down by the learning rate α, to the current policy parameters.

Algorithm 6.2 Gradient Biasing

Input: Transitions T, Current policy parameters θ_k, Solved policy parameters $\theta^*_{M,k}$, Biasing rate λ_k, Learning rate α

Output: Updated policy parameters θ_{k+1}

1: $\nabla_\theta J(\theta)|_{\theta=\theta_k} \leftarrow VANILLA\text{-}POLICY\text{-}GRADIENT(\theta_k, T)$
2: $\nabla_\theta J(\theta)^+|_{\theta=\theta_k} \leftarrow \nabla_\theta J(\theta)|_{\theta=\theta_k} + \lambda_k(\theta^*_{M,k} - \theta_k)$ // Biasing step
3: $\theta_{k+1} \leftarrow \theta_k + \alpha\nabla_\theta J(\theta)^+|_{\theta=\theta_k}$
4: Return θ_{k+1}

To our knowledge, this is the first algorithm capable of directly incorporating model knowledge in policy gradient updates for general MDPs. Wang and Dietterich [25] published a policy gradient algorithm based on partial transition and reward models, but they were limited to MDPs where all policies are proper, so every episode terminates.

6.2.4 Biasing Rate

As we mentioned in the Section 6.2.2, the biasing rate λ_k is a free parameter in this algorithm. λ_k represents how much weight we give the past experience models as opposed to the observations we have seen in the current MDP. If λ_k is high, then we rely more heavily on the models, and if λ_k is low, then we rely more heavily on the current observations. Our approach to setting λ_k is to initialize λ_0 to a high value and then decay it as an exponential function of k:

$$\lambda_k \leftarrow w_{input} e^{-(k-1)w_{decay}} \tag{6.3}$$

w_{input} and w_{decay} are tuneable constants. w_{input} represents the maximum weight that the biasing policy parameters are given relative to the vanilla policy gradient. w_{decay} represents the rate that the decay occurs. This formulation lets us quickly move to the optimal model policy at the start of the task, but then steadily give more weight to what we have observed in the current environment.

Decaying λ_k is important because our models are not perfect and the model that we select may not match the current environment very well. In that case, once the current observations become the dominant factor in the parameter update, we will still be able to find a good policy.

One disadvantage to this strategy is that as we gain more experience in the current environment, we are more likely to make a good choice among the available models. We have therefore decided to reset λ_k to λ_0 every time the choice of models changes.

6.3 Knowledge Structuring

An interesting question from the point of view of long-lived RL agents is how they manage their knowledge. This knowledge needs to be structured at two different levels: within individual domains and across all domains. These two aspects inform our approach: we use tree-structured models within a domain and keep track of a fixed number of different domains at a time. We covered the tree-structured models in Section 6.3.1 and the different domains in 6.3.2. Our approach can be viewed as a weaker version of the Bayesian modeling that we discussed in Section 2.7.4.

6.3.1 Matching

Recall the underlying distribution ζ from Chapter 3. ζ can select the transition and reward parameters independently from each other. This implies that we need de-coupled transition and reward models. The transition models need to map (s, a)

pairs to s' outcome states, and the reward models need to map (s, a) pairs to r reward values.

We also previously mentioned the need for the models to be able to fit arbitrary decision functions. This brings us back to the concept of model trees and, specifically, the linear Gaussian trees from Section 4.4.1. As we saw in that section, linear Gaussian trees allow us to find the probability of an output given an input. We can then find the probability of an s' given an (s, a) pair for a transition tree and an r given an (s, a) pair for a reward tree. To find the best trees for the transitions $T_{1..n}$ of the current environment, we can calculate a probability score ϵ for each transition tree:

$$\epsilon_{tree} = \sum_{i=1}^{n} \log P(s'_i | s_i, a_i) \tag{6.4}$$

and similarly for each reward tree, using r_i rather than s'_i. We create the best model world W by selecting the transition tree with the highest score and the reward tree with the highest score.

Algorithm 6.3 gives the full matching procedure. The agent selects the most similar transition and reward models (Steps 1 and 2) to the current environment. The combination of the two models creates a model-world W. It then takes the samples associated with the chosen transition model (Step 3) and passes them through the reward models to generate predictions r for each of the (s, a) pairs (Step 4). The r predictions, along with the s' outcome states, create (s, a, s', r) full transitions from W. One assumption that we have made here is that the closest W is a reasonable approximation of the current MDP, and the optimal policy parameters for W will usually be a good initial guide for the current policy parameters. An alternative would be to not select any model, if no model was close enough to the transitions observed so far, and simply run a non-augmented forward policy gradient algorithm. However, this introduces an additional necessary free parameter: the threshold value

Algorithm 6.3 Matching

Input: Transitions $(s, a, s', r)_{1..n}$, Transition Models M_Δ, Samples array A, Reward Models M_R

Output: (s, a, s', r) samples, from best models for transitions

1: $\textit{best-transition-index} \leftarrow \arg\max_{tree \in M_\Delta} \epsilon_{tree}((s, a, s')_{1..n})$
2: $\textit{best-reward-tree} \leftarrow \arg\max_{tree \in M_R} \epsilon_{tree}((s, a, r)_{1..n})$
3: $\textit{best-transitions} \leftarrow A[\textit{best-transition-index}]$
4: $\textit{transition-rewards} \leftarrow \textit{best-reward-tree}(\textit{best-transitions})$
5: Return $\textit{best-transitions} \times \textit{transition-rewards}$ // Join the saved samples with rewards from the reward model

for ϵ that creates the boundary between using a model and not using a model. Our approach eliminates the need for this parameter. If the distributions ζ that we use in our experiments produced many MDPs with sufficiently different optimal policies, it might be desirable to establish a cutoff threshold, instead. Fortunately, even if counter-productive policy parameters are returned by the models, our forward RL strategy should be able to eventually recover due to the decay in the biasing rate λ_k, discussed in Section 6.2.4.

6.3.2 Model Lists

We have mentioned that we keep a list of transition models, samples associated with them, and a list of reward models. The update process for the lists is given in Algorithm 6.4. We take as a parameter the maximum number of models per list to avoid unbounded memory usage. During execution on an environment, we do not make any changes to the list or samples. After environment execution we create new transition and reward trees from the (s, a, s', r) transitions seen from the environment (Step 1). If the lists are not yet full, we add them to the lists along with the transitions (Step 3). If the lists are full, we rank the existing trees by similarity to the new environment as in Equation (6.4) (Step 5). Then we evict the most similar transition and reward trees and put the new trees in their place along with the new transitions

(Step 6).

Algorithm 6.4 Update Model Lists

Input: Transition or reward model list M_L, new transitions or reward samples S, max list size $L_{\max}$

1: $M_{new} \leftarrow$ CREATE-MODEL(S)
2: **if** $M_L.size < L_{\max}$ **then**
3: $\quad M_L.append(M_{new})$
4: **else**
5: $\quad$ *best-index* $\leftarrow$ *best-model-index*(M_L, S) // Either Step 1 or Step 2 of Algorithm 6.3
6: $\quad M_L[$*best-index*$] \leftarrow M_{new}$ // Evict the closest model
7: **end if**

Removing the most similar tree encourages diversity of the models. Even if some regions of ζ are observed less frequently, we would like to keep their models available. For the more dense regions, a small number of models representing that region should suffice, since the optimal policies in a region should be similar. This method of updating the lists maintains that balance.

Each model can be thought of as representing a region of the distribution ζ. ζ can be a continuous function and the number of models is finite, which means that the models perform some implicit discretization over ζ. However, unlike most discretization methods which require either human intervention to define bin boundaries or a uniform grid that ignores the specifics of the MDP, the model creation process makes the break points at natural boundaries based on the actual environments encountered.

6.4 Optimizations for Fixed Reward Functions

When the reward function does not change, the model updates and usage can be simplified. There is no longer a need for a list of reward models. In fact, there is no need for any reward models. After termination of an environment, we create a tree model for its (s, a, s') transitions. Then we take the best policy parameters

found during the environment execution and store them along with the transition model. During execution of an environment, when the agent requests the best policy parameters, the transitions for the environment are again scored on each tree as in Equation (6.4). The most likely tree is identified and its stored solved policy parameters are passed to the agent.

These simplifications result in reduced memory footprint (because samples no longer need to be stored) and speed improvements (because reward trees no longer need to be created and scored). It does, though, come at the cost of flexibility on the reward functions.

6.5 Policy Compatibility

We now know how to use our models to find a good policy for the current environment. In Section 6.2, we described how we use those policies to improve the current policy through gradient biasing. However, there is one intermediate step that must be taken between FQI and gradient biasing. The output of FQI as presented in Algorithm 5.4 is a set of (s, a) pairs representing the policy. In order for the biasing to work, we need a policy with the same basis ϕ as the policy that our online learner uses.

Algorithm 6.5 details how we get a set of policy parameters θ that match those of the target policy. In Step 1, we take all the states that we know of (the pre- and post-transition states from the input samples) and find their values under $\phi(s)$. Then in Step 2 we do a linear regression from those values to the actions. This produces the closest compatible θ approximation of the optimal FQI policy parameterizable by $\phi(s)$.

Algorithm 6.5 Linearize Policy

Input: Basis function ϕ, Sample states $s_{1..n}$ with actions $a_{1..n}$ according to the FQI policy

Output: Policy parameters θ

1: $s_{\phi,1..n} \leftarrow \phi(s_{1..n})$ // Convert the states to the basis form
2: $\theta \leftarrow LINREGRESS(s_{\phi,1..n}, a_{1..n})$ // Find the weights that best linearize the FQI policy under the basis
3: **return** θ

6.6 Experiments

6.6.1 Augmented Deterministic Policy Gradient

The model incorporation method works for any policy gradient algorithms. In order to be able to run experiments with it, though, we need to choose an algorithm to which we can actually apply it. We have chosen to do this with Silver et. al's Deterministic Policy Gradient algorithm family [8], which we described in Section 2.7.1, because it seems to have excellent performance on a wide variety of MDPs. The equations are unchanged, except for the update for the actor basis weights, which was:

$$\theta_{k+1} \leftarrow \theta_k + \alpha_\theta \nabla_\theta \mu_\theta(s_k)(\nabla_\theta \mu_\theta(s_k)^\top w_k) \tag{6.5}$$

The new update equation is:

$$\theta_{k+1} \leftarrow \theta_k + \alpha_\theta \nabla_\theta \mu_\theta(s_k)(\nabla_\theta \mu_\theta(s_k)^\top w_k) + \lambda_k(\theta^*_{M,k} - \theta_k) \tag{6.6}$$

Applying the gradient augmentation is easy. This can be repeated for other policy gradient algorithms as well.

6.6.2 Hypotheses

There are two different tracks to investigate for this chapter: matching and gradient biasing.

For matching, we hypothesize that our tree models will be able to distinguish environments with different transition and reward functions. As the models become weaker, with lower tree depth and fewer parameters available to them, the ability to distinguish between environments will degrade.

For gradient biasing, we hypothesize that biasing will improve learning if the biasing policy was trained on an environment similar to the current one. In the case where it was trained on an environment dissimilar from the current one, our forward learners will eventually be able to recover due to the decay of λ.

6.6.3 Methodology

Simpler models have a tendency to consistently report higher probabilities than others. We accounted for this by performing baseline adjustment, which means that when a model is trained we record the average log odds that it reports on its training examples. When new samples are tested for matching, we subtract the average score from the output. All of our matching results are reported as baseline-adjusted average log odds per sample.

Sometimes, our weak models are restricted to fewer variables than the number of inputs (the number of state features plus action variables). In this case we choose a combination of inputs by testing all possible combinations of the allowed number of variables and choosing the combination with the best performance.

For gradient biasing, we used Deterministic Policy Gradient with the same parameters as given in Section 4.5.3. Additionally, we used model input rate parameters (from Section 6.2.4) of $w_{input} = 0.001$ and $w_{decay} = 0.001$.

6.6.4 Results and Discussion

We ran arm reward matching with 5000 samples to train the models and 2000 samples to match with. We used samples from three different IRL rewards, with the targets at distances 0.1017, 0.1081, and 0.0726 meters from the starting position. The trees were allowed to grow to depth 3. Table 6.1 shows the results of this experiment.

	0.1017-Goal Tree	0.1081-Goal Tree	0.0726-Goal Tree
0.1017-Goal Samples	0.1257	0.0163	0.1225
0.1081-Goal Samples	-0.2483	0.1482	-0.1651
0.0726-Goal Samples	0.0236	0.0406	0.2123

Table 6.1: Matching results for arm reward trees with depth 3. Table entries are baseline-adjusted average log odds per sample.

The reward matching works well. In all three cases, the trees are able to match the correct reward function.

We ran arm transition matching with 2000 samples to train the models and 2000 samples to match with. We used samples from environments with three different resistance values: 0.0, 1.5, and 3.0. We ran experiments with three different model strengths: trees with depth 3, in Table 6.2; linear Gaussians with all three parameters (position, velocity, and force) available, in Table 6.3; and linear Gaussians with only two parameters available, in Table 6.4. In the last scenario, the model was forced to choose only the two most predictive parameters to use.

The transition matching also works well, for the first two model strengths. When we move from linear Gaussians with 3 parameters to linear Gaussians with 2 parameters, though, the matching results are almost meaningless. This suggests that a linear Gaussian model is sufficient, but the transition function relies on all parameters to

	0-Resistance Tree	1.5-Resistance Tree	3.0-Resistance Tree
0-Resistance Samples	0.10	-40.53	-41.64
1.5-Resistance Samples	-41.06	0.13	-41.76
3.0-Resistance Samples	-42.01	-42.35	0.03

Table 6.2: Matching results for arm transition trees with depth 3. Table entries are baseline-adjusted average log odds per sample.

	0-Resistance LG	1.5-Resistance LG	3.0-Resistance LG
0-Resistance Samples	-0.24	-40.63	-41.65
1.5-Resistance Samples	-39.49	0.00	-41.76
3.0-Resistance Samples	-40.48	-42.46	0.01

Table 6.3: Matching results for arm transition weak model linear Gaussians, with number of input variables restricted to 3. Table entries are baseline-adjusted average log odds per sample.

update the state. This is in line with the equations that we gave for the arm simulator in Section 3.3.5.

For gradient biasing, we ran three learners on the arm simulator with 3.0 resistance and target of 0.0792 meters from the starting state. Each of them was biased with a policy trained on 3.0 resistance and one of the available targets: 0.0726 (close to the current environment); 0.0528 (medium distance from the current environment); and 0.1017 (far from the current environment). This is intended to test the performance of the algorithm both when it has information that should be helpful and when it has to recover from being biased with a policy that does not perform well on the current environment.

	0-Resistance LG	1.5-Resistance LG	3.0-Resistance LG
0-Resistance Samples	0.04	-0.47	-0.76
1.5-Resistance Samples	0.33	0.09	-0.08
3.0-Resistance Samples	0.48	0.13	0.21

Table 6.4: Matching results for arm transition weak model linear Gaussians, with number of input variables restricted to 2. Table entries are baseline-adjusted average log odds per sample.

We also ran a non-biased learner for comparison.

We ran each of the scenarios three different times and averaged the results. Each run went for 200 cycles, interspersing a single learning episode with 10 evaluation episodes from random starting states. Results are shown in Figure 6.1.

The results in Figure 6.1 show that biasing with a policy from a similar environment (the 0.0726 target) starts out about as well as the no-biasing case and then improves much more quickly. Since the biasing policy in this case is not perfect, and influences the agent to move to a position that is close to the goal but not close enough to end the episode, it makes sense that the initial reward is low. In subsequent episodes, it is able to learn to move to the 0.0792 target rather than the 0.0792 target, as well as taking advantage of the previous knowledge in the rest of the state space that allows it to efficiently move towards the correct region. This shows that we can significantly improve learning when we have a policy available from a similar environment.

Biasing with a policy a medium distance from the current environment (the 0.0528 target) starts worse than the no-biasing policy but then quickly catches up. In the first few episodes, the agent rarely reaches the goal because the 0.0528 target is not close to the 0.0792 target. Once the agent does learn to move to the correct target,

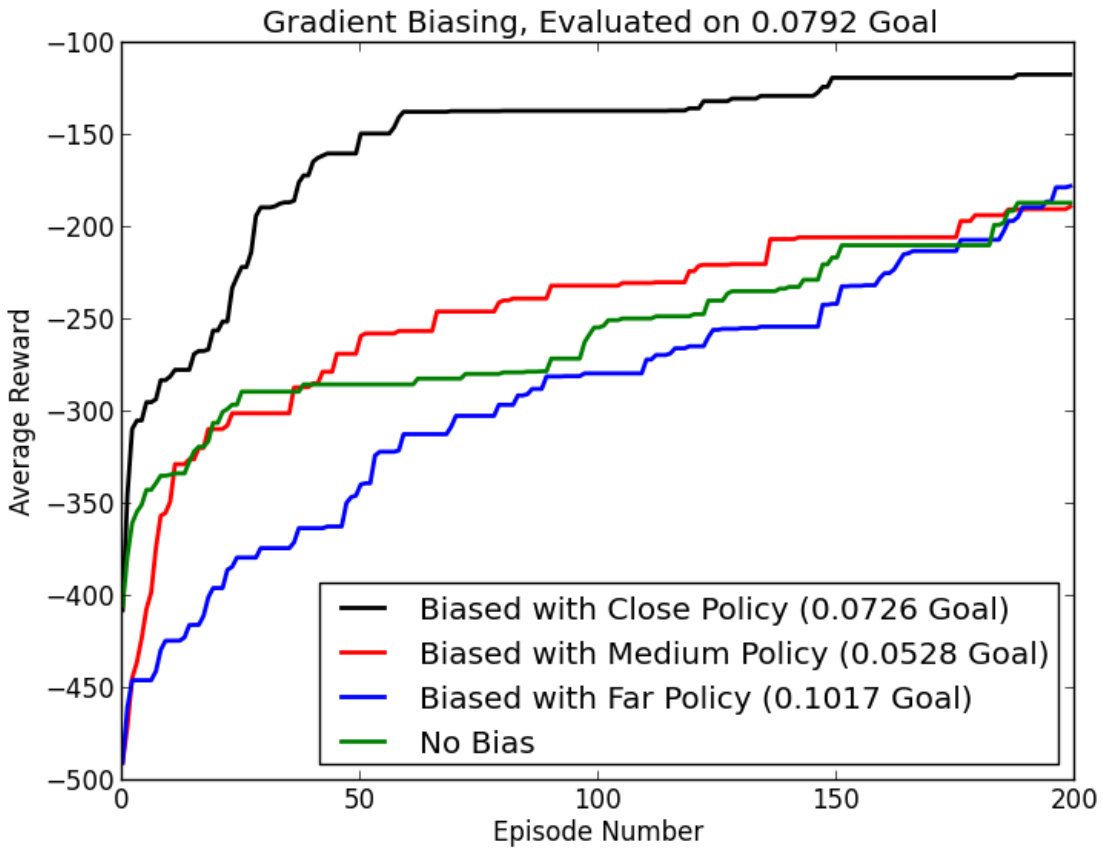

Figure 6.1: Improvement on arm forward RL with gradient biasing, goal of 0.0792

learning proceeds unhindered.

Finally, biasing with a policy a far distance from the current environment (the 0.1017 target) starts worse than the no-biasing policy and remains so for many episodes. Eventually, it is able to reach parity with the no-biasing policy. This shows the effectiveness of our decay parameters. It also highlights the danger of negative transfer: if we are using a policy that is not well suited to the current environment, then it can have a negative impact on learning.

Chapter 7

Empirical Evaluation of Tree-Structured Continuous MTRL

This chapter presents results for the combined pipeline created in the previous three chapters.

7.1 Hypotheses

We hypothesize that using our pipeline will improve the sample efficiency of a sequence of tasks. That is, when we encounter a new task and already have an established knowledge base, we will be able to find a good policy more quickly when measured by the number of samples and episodes required. If the prior environments do not include an appropriate policy then the learner will either choose not to perform biasing or recover quickly from the biasing.

We additionally hypothesize that using our pipeline will improve the ability of the learner to perform the task with the same starting state as the expert demonstrations, rather than random starting states.

7.2 Methodology

As we saw in Chapter 6, using policies from environments that are too dissimilar from the current one can reduce performance.

We added a mechanism to avoid biasing if none of the previous environments are a good enough match. When the model is created, we sort the log probabilities across all input values. Then we select a value, p_{reject}, that is N_{reject} percent of the way through the sorted array. N_{reject} is a tuneable parameter. When a new set of samples is tested on the model, we do not bias if the median log probability on the new set of samples is lower than p_{reject}. For these experiments, N_{reject} was set to 30%.

Our choice for the $SHOULD\text{-}RUN\text{-}PROCEDURE(k)$ function from Algorithm 6.1 is to run the matching procedure if the step number k is a power of 2. This allows us to initially run the matching often as we gain more experience about the environment, and then spend less time on it later as our estimates of the best models stabilize.

We tested on several different environment configurations. We ran each one learner on each environment for 100 cycles, interleaving one learning cycle with 10 evaluation cycles.

7.3 Results and Discussion

Our first experiment, in Figure 7.1, tests the combined matching and biasing procedures when previous knowledge from a similar environment is available. We ran a learner on resistance 3.0 and target 0.0792 for 500 cycles. We saved the best policy and a reward model with tree depth 3, trained on 1000 samples. Then we used that information for a new set of 5 learners with resistance 3.0 and target 0.0726 for 100 cycles. We allowed the new learners to accept or reject biasing based on the matching results. For comparison, we ran 5 learners in the same scenario with no previous

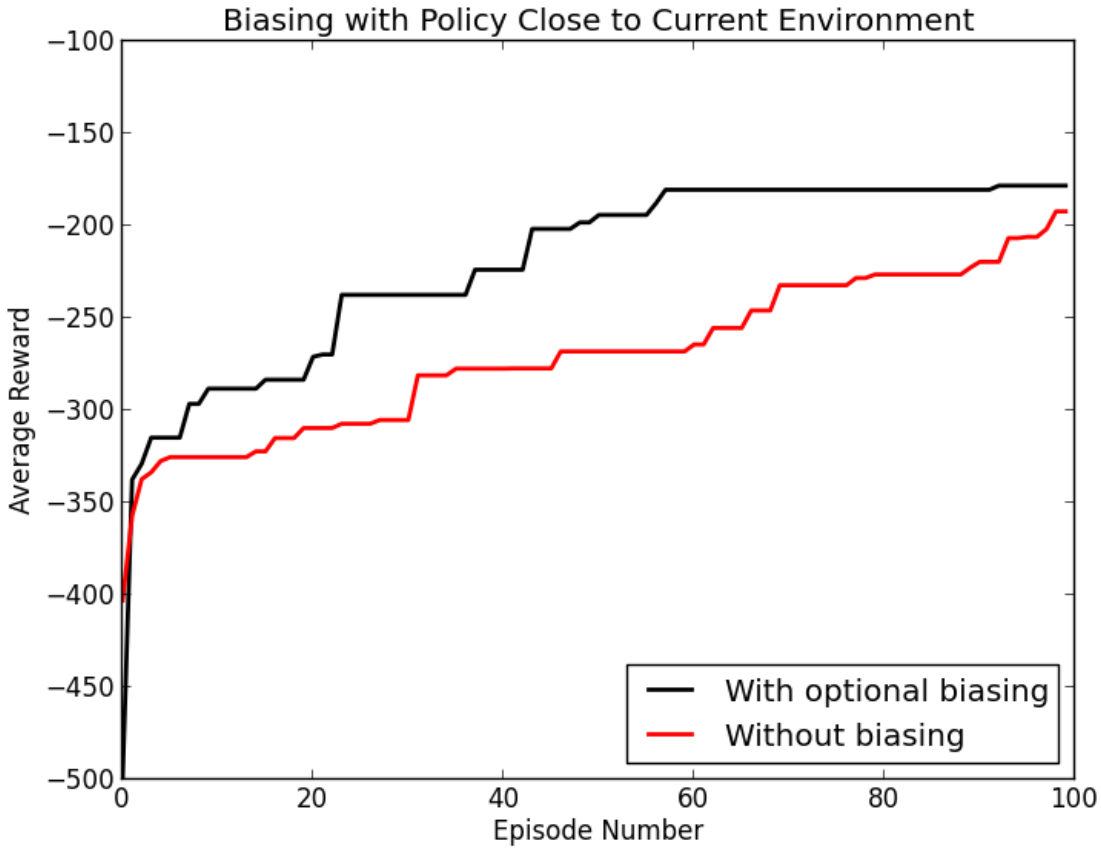

Figure 7.1: Learners with a target of 0.0726 were given the option to bias with knowledge from an environment with target 0.0792.

knowledge available. The agent chose to perform biasing at most steps where it was available. The results show improvement when biasing is available, except for the very first episode. This is because the biasing influences the agent to move to the 0.0792 target and then stop, so it does not reach the 0.0726 region. Subsequently, it learns that this strategy does not produce good rewards, and is able to take advantage of the other, more useful knowledge that it is given. This shows that the full pipeline can be very useful when relevant knowledge is available.

The second case had the same experimental setup, but the knowledge came from an environment with a target of 0.0528. Results are shown in Figure 7.2.

The agent chose to perform biasing at approximately half the steps where it was

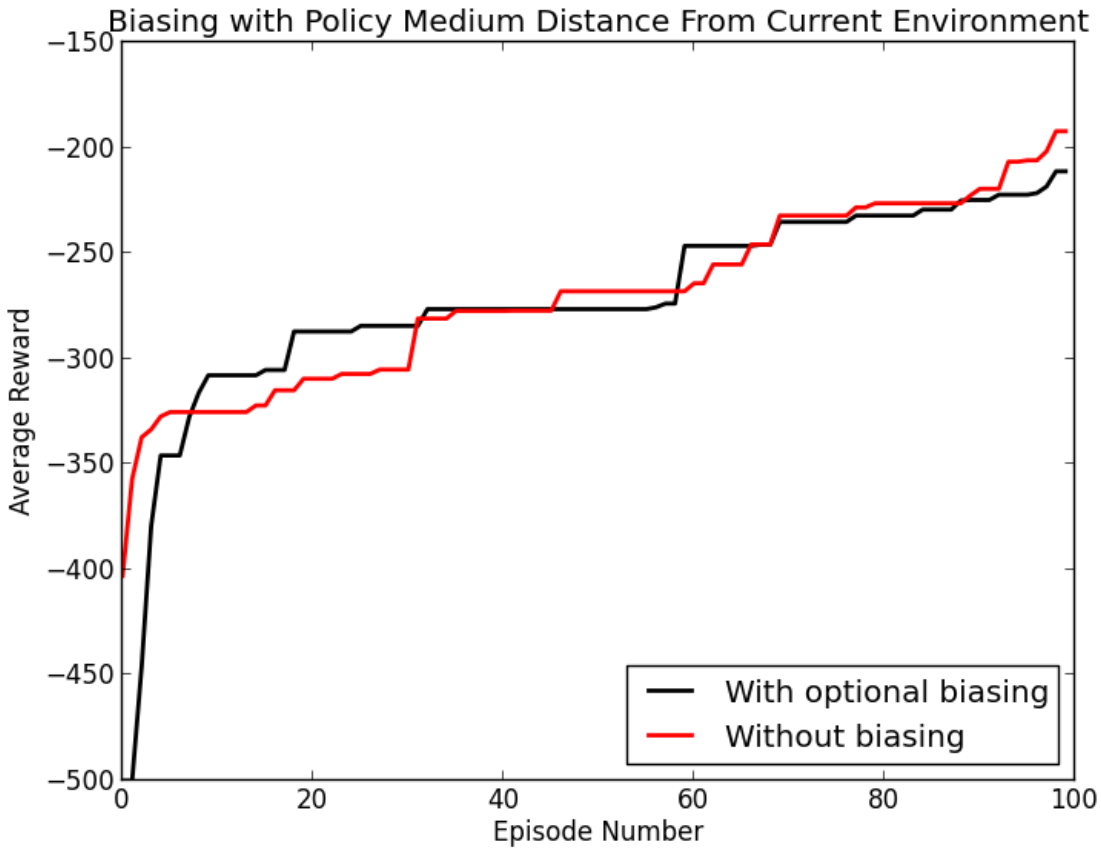

Figure 7.2: Learners with a target of 0.0726 were given the option to bias with knowledge from an environment with target 0.0528.

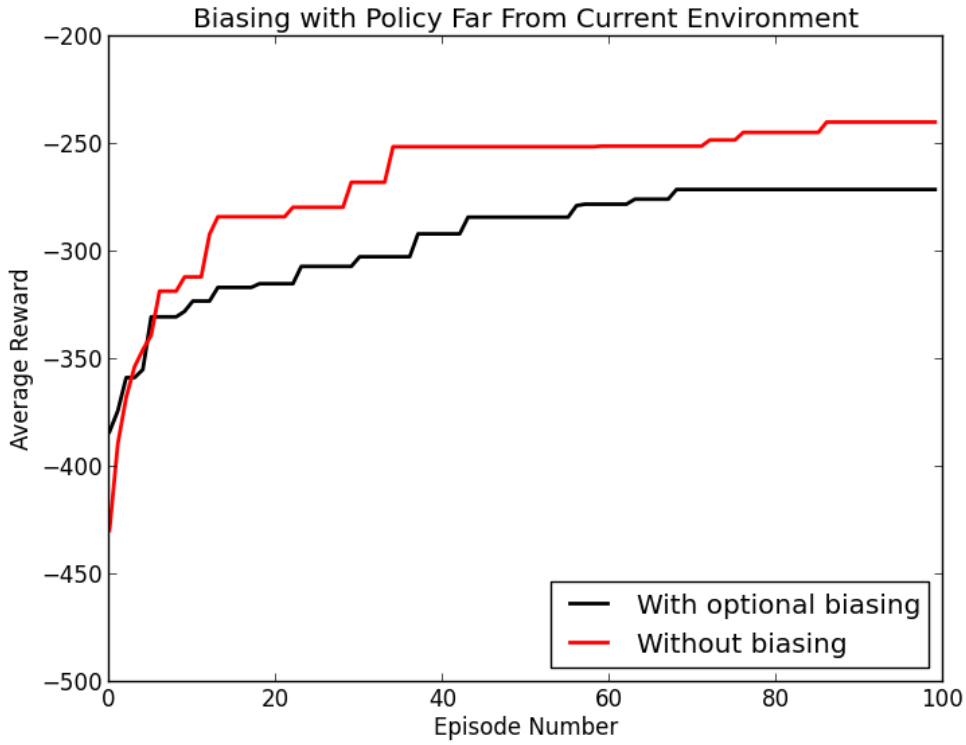

Figure 7.3: Learners with a target of 0.0528 were given the option to bias with knowledge from an environment with target 0.1017.

available. The results show that initially, as in the previous experiment, the biased learner underperforms the non-biased learner. After a few episodes, it is able to catch up and does equally well through the rest of the learning. This shows that even when the available knowledge is from an environment that is not similar to the current one, our system does not see a significant negative impact.

In the third case, we obtained models and policies from an environment with a target of 0.1017, and then used those on an environment with target 0.0528. Results are shown in Figure 7.3.

The biased learner chooses to reject biasing most of the time. However, the non-biased learner still generally does better than the biased learner. This is because even when the biased learner rejects biasing most of the time, it still chooses to bias on a few steps. Because we reset the decay λ when the chosen biasing policy changes, the

most impactful biasing steps are applied on the rare occasions that it does choose to bias. This negatively impacts the policy. However, it is still able to mostly recover from the negative transfer and shows clear improvement over time.

Our final experiment was to run two learners with resistance 0 and target 0.1017. The first learner had model and policy knowledge from a previous learner that was trained with resistance 0 and target 0.1081. The second learner had no prior knowledge. For these learners, similarly to the last set of experiments in Chapter 4, we started both the learning and evaluation episodes at the zero state (position 0, velocity 0). This is the same configuration as the real data was obtained with. This is different from the random starting state that we used previously, which could have started at any combination of legal positions and velocities.

Results are shown in Figure 7.4. We ran the learners for 300 learning episodes each, and saved the policies which reached the goal fastest during evaluation.

The augmented learner first reached the goal after 13 learning episodes. The non-augmented learner first reached the goal after 113 learning episodes. The augmented learner also reaches the goal more quickly than the non-augmented learner in the best policies.

This shows that our method of knowledge transfer improves learning on the task which we originally set out to perform. It lets us learn about one environment and then use that knowledge to learn more quickly on a new, similar environment.

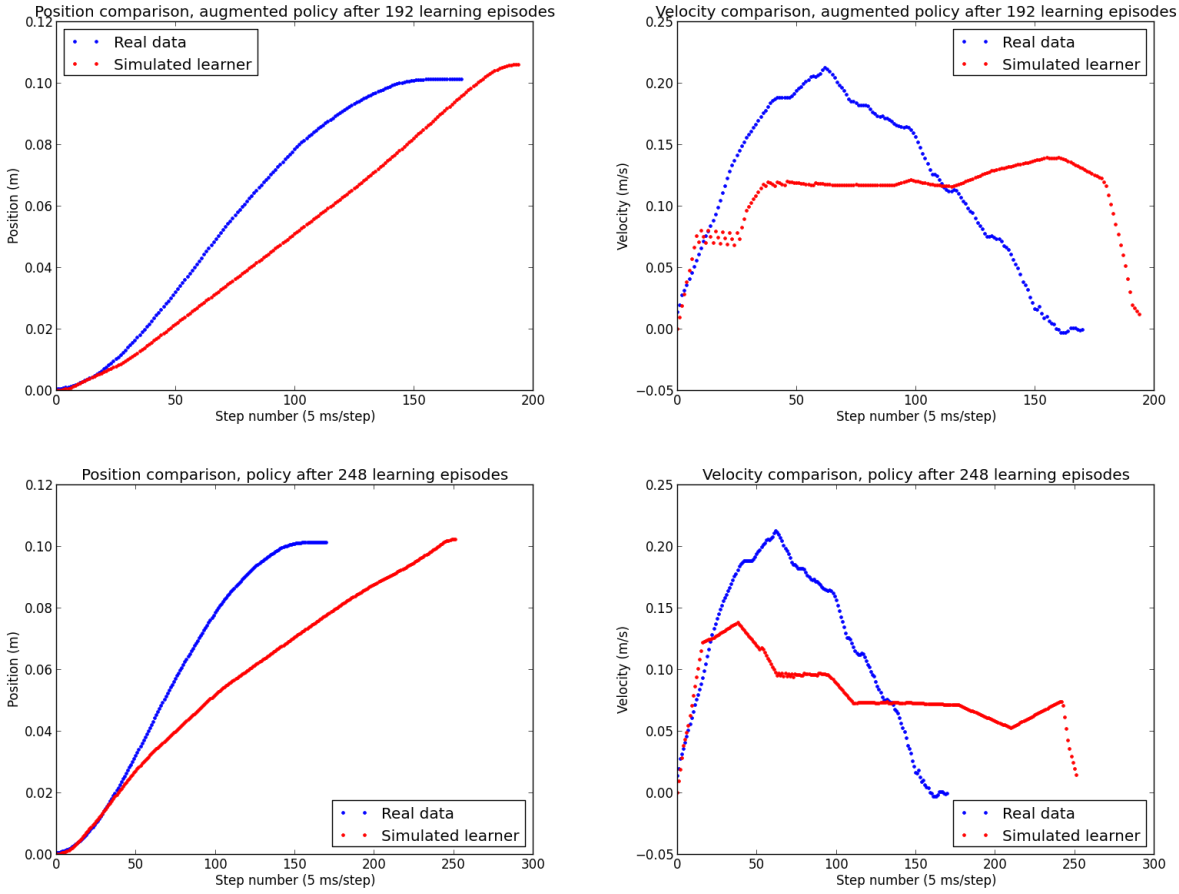

Figure 7.4: Position and velocity trajectories for the best learner over 300 episodes starting from the zero state. The two graphs in the top row are augmented, and the two graphs in the bottom row are not. The red curves are the simulated learner and the blue curves are the real data.

Chapter 8

Conclusion

We started this thesis with the problem of prosthetic arm motion. We wanted to learn how arms usually move by observing demonstrations. We then used that information to find policies that could reproduce the movement intention. Since the scenarios under which we collected data varied, we could not learn a single policy that worked well for all of them. Rather than learning in each scenario individually, we addressed the problem of knowledge transfer. Each previous scenario taught us something about the arm domain in general, and we wanted to use that prior knowledge to speed up learning on future tasks.

Our contributions covered three main areas: IRL and offline RL with continuous actions, non-smooth reward functions, and learning with weak models on a sequence of tasks.

We addressed continuous actions in every chapter. We explained why it is useful for our policies, Q-functions, and reward representations to process actions over a continuous space rather than discretizing them. We developed continuous-action versions of two existing discrete-action algorithms, CSIRL and FQI. This is directly applicable to our motivating problem of prosthetic arm motion since we move in continuous spaces with continuous actions and prostheses should reflect that.

Our approach to non-smooth reward functions was to use trees. Again, trees appeared in every chapter. We presented a tree-based version of CSIRL that was able to reproduce the goal-oriented behavior of the arm problem. Our Continuous FQI algorithm used trees in order to handle continuous actions, but this also had a great benefit to the representation quality of the intermediate Q-functions.

We developed a novel addition to any policy gradient algorithm that can take advantage of information from known policies as well as adapting to a new environment. This gave us a mechanism to incorporate transferred knowledge from previous tasks into a new task. Additionally, we designed an algorithm that could identify similarities between the transition and reward functions for environments. It was able to use representations of arbitrary compactness that improved as more information was available.

We ran our experiments on interesting domains, but they were low-dimensional. We would like to explore how our algorithms scale to higher dimensions with more state variables and potentially more actions. This could require finding opportunities for performance improvements, particularly in tree construction.

Weak models are another point of interest. We showed that we could do matching with a limited amount of information. We would also like to find a way to eliminate the requirement of storing large numbers of samples or large tabular policy representations. This could bring us closer to a knowledge transfer mode that approximates on one level how the human brain works.

We noted in Section 4.5.4 that testing each unique possible split value is the main computational cost in creating regression trees. For decision trees, there are established techniques to reduce the number of splits that need to be considered, but we did not find an analogous optimization when the label space is continuous. This would be an interesting direction for future exploration.

Bibliography

[1] G. M. Nelson, R. D. Quinn, R. J. Bachmann, W. Flannigan, R. E. Ritzmann, and J. T. Watson, "Design and simulation of a cockroach-like hexapod robot," in *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, vol. 2. IEEE, 1997, pp. 1106–1111.

[2] R. Bellman, "Dynamic programming and lagrange multipliers," *Proceedings of the National Academy of Sciences*, vol. 42, no. 10, pp. 767–769, 1956.

[3] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[4] M. Sugiyama, *Statistical Reinforcement Learning: Modern Machine Learning Approaches*. CRC Press, 2015.

[5] M. G. Lagoudakis and R. Parr, "Least-squares policy iteration," *Journal of machine learning research*, vol. 4, no. Dec, pp. 1107–1149, 2003.

[6] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[7] J. R. Quinlan *et al.*, "Learning with continuous classes," in *5th Australian joint conference on artificial intelligence*, vol. 92. Singapore, 1992, pp. 343–348.

[8] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *ICML*, 2014.

[9] P. S. Thomas, "A reinforcement learning controller for functional electrical stimulation of a human arm," Ph.D. dissertation, Case Western Reserve University, 2009.

[10] F. Fernández and M. Veloso, "Exploration and policy reuse," CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, Tech. Rep., 2005.

[11] F. Fernández and M. M. Veloso, "Reusing and building a policy library." in *ICAPS*, 2006, pp. 378–381.

[12] R. Dearden, N. Friedman, and D. Andre, "Model based bayesian exploration," in *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1999, pp. 150–159.

[13] A. Wilson, A. Fern, S. Ray, and P. Tadepalli, "Multi-task reinforcement learning: a hierarchical bayesian approach," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 1015–1022.

[14] P. Abbeel, A. Coates, and A. Y. Ng, "Autonomous helicopter aerobatics through apprenticeship learning," *The International Journal of Robotics Research*, vol. 29, no. 13, pp. 1608–1639, 2010.

[15] I. S. MacKenzie, "Fitts' law as a research and design tool in human-computer interaction," *Human-computer interaction*, vol. 7, no. 1, pp. 91–139, 1992.

[16] M. J. Fu, "Computational models and analyses of human motor performance in haptic manipulation," Ph.D. dissertation, Case Western Reserve University, 2011.

[17] M. C. Cavusoglu and D. Feygin, "Kinematics and dynamics of phantom (tm) model 1.5 haptic interface," *University of California at Berkeley, Electronics Research Laboratory memo M*, vol. 1, 2001.

[18] E. Klein, B. Piot, M. Geist, and O. Pietquin, "A cascaded supervised learning approach to inverse reinforcement learning," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2013, pp. 1–16.

[19] F. Provost and P. Domingos, "Well-trained pets: Improving probability estimation trees," 2000.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[21] D. Ernst, P. Geurts, and L. Wehenkel, "Tree-based batch mode reinforcement learning," *Journal of Machine Learning Research*, vol. 6, no. Apr, pp. 503–556, 2005.

[22] A. Antos, C. Szepesvári, and R. Munos, "Fitted q-iteration in continuous action-space mdps," in *Advances in neural information processing systems*, 2008, pp. 9–16.

[23] L. Busoniu, R. Babuska, B. De Schutter, and D. Ernst, *Reinforcement learning and dynamic programming using function approximators*. CRC press, 2010, vol. 39.

[24] N. Parikh, S. Boyd *et al.*, "Proximal algorithms," *Foundations and Trends® in Optimization*, vol. 1, no. 3, pp. 127–239, 2014.

[25] X. Wang and T. G. Dietterich, "Model-based policy gradient reinforcement learning," in *ICML*, 2003, pp. 776–783.

[26] R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour *et al.*, "Policy gradient methods for reinforcement learning with function approximation." in *NIPS*, vol. 99, 1999, pp. 1057–1063.

[27] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[28] N. Aghasadeghi and T. Bretl, "Maximum entropy inverse reinforcement learning in continuous state spaces with path integrals," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on.* IEEE, 2011, pp. 1561–1566.

[29] M. J. Fu and M. C. Cavusoglu, "Human-arm-and-hand-dynamic model with variability analyses for a stylus-based haptic interface," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 42, no. 6, pp. 1633–1644, 2012.

[30] M. J. Fu and M. C. Æavuşoğlu, "Three-dimensional human arm and hand dynamics and variability model for a stylus-based haptic interface," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on.* IEEE, 2010, pp. 1339–1346.