Adaptive Representation for Policy Gradient

by

Ujjwal Das Gupta

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science
in
Statistical Machine Learning

Department of Computing Science
University of Alberta

Abstract

Much of the focus on finding good representations in reinforcement learning has been on learning complex non-linear predictors of value. Methods like policy gradient, that do not learn a value function and instead directly represent policy, often need fewer parameters to learn good policies. However, they typically employ a fixed parametric representation that may not be sufficient for complex domains. This thesis introduces two algorithms which can learn an adaptive representation of policy: the Policy Tree algorithm, which learns a decision tree over different instantiations of a base policy, and the Policy Conjunction algorithm, which adds conjunctive features to any base policy that uses a linear feature representation. In both of these algorithms, policy gradient is used to grow the representation in a way that enables the maximum local increase in the expected return of the policy. Experiments show that these algorithms can choose genuinely helpful splits or features, and significantly improve upon the commonly used linear Gibbs softmax policy, which is chosen as the base policy.

Acknowledgements

I want to thank my supervisors, Dr. Michael Bowling and Dr. Erik Talvitie, working with whom has been an amazing experience. I am very grateful to have had the freedom to work on problems that I found interesting, along with their support and guidance whenever I needed it.

I also thank all the friends I have made at the University of Alberta, especially Sriram, Ankush, Talat and Shiva. To paraphrase one of my favourite writers, "summer friends may melt away like summer snows, but winter friends are friends forever."

Finally, I thank my parents. Their love and support keeps me going on.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> It is pointless to do with more what can be done with fewer.
>
> WILLIAM OF OCKHAM

An intelligent agent is defined to be an entity which can perceive its environment through sensors and act upon it using actuators, with the aim of achieving a certain goal. Reinforcement learning is concerned with creating such agents, where the goal is to maximize some notion of cumulative reward obtained. These agents learn about the environment by interacting with it, and construct a mapping from states to actions, which is known as the policy. Several problems, such as learning to perform aerobatic manoeuvres with a helicopter (Ng et al., 2006), or learning to play games (Bellemare et al., 2013), can be cast as reinforcement learning problems.

Reinforcement learning algorithms are typically of two types. The first are value function based methods, which aim to learn an accurate function mapping from states to values, from which a policy can be obtained. The value function represents an estimate of the cumulative reward that can be obtained by taking an action in a state, which indicates the benefit of choosing the action. The second are policy search methods, which directly learn a function from states to policies. In both of these techniques, the search for state representation is an important and challenging problem. The representation should allow generalization of learned values or policies to unseen states, and at the same time, needs to be powerful enough to represent sufficiently complex functions.

In value based algorithms, a typical way to achieve generalization is to approximate the value function using a linear function over the features associated with the state. However, these methods are not guaranteed to work well when function approximation is used (Boyan and Moore, 1995), and so learning an accurate repre-

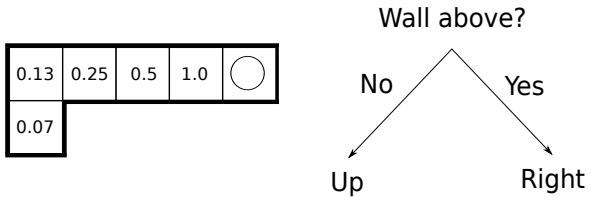

Figure 1.1: A simple world, and a corresponding policy tree

sentation of state is important. Also, value based methods produce a deterministic policy, which is not guaranteed to be optimal when the state does not encapsulate all relevant information about the environment (Singh et al., 1994). A way to get an adaptive representation of state which can deal with these problems is to learn a decision tree over the history of observations. The U-Tree algorithm (McCallum, 1996) is an example of this. It starts with a single node to represent state, and recursively performs statistical tests to check if there exists a split for which the child nodes have significantly different value. The resulting tree represents a piecewise constant estimation of value. Some enhancements to U-Tree include alternative heuristics for growing the decision tree (Au and Maire, 2004), and extensions to continuous (Uther and Veloso, 1998) and relational (Dabney and McGovern, 2007) domains.

Such an algorithm would grow the tree whenever doing so improves the value estimate. It is possible for an entire branch of the tree to contain multiple splits over fine distinctions of value, even if the optimum action from each of the nodes in the branch is the same. For example, Figure 1.1 shows a simple grid world with a single terminal state, indicated by a circle, transitioning to which gives a reward of 1. The agent is equipped with four sensors to detect whether there is a wall in each direction. Assuming a discount factor of 0.5, the values of the states are shown in each grid cell. To represent an accurate value function, a decision tree would need to split on current as well as past observations, and eventually distinguish between all the states. In complex domains, one would end up learning a very large tree. In contrast, if one were to represent the policy itself as a function over the observations, a simple and optimal policy can be obtained in the form of the decision tree on the right.

Policy gradient algorithms are an alternate approach to reinforcement learning, which directly learn a function from states to actions. The function is optimized by

the principle of stochastic gradient ascent. Unlike value-based methods, they can represent stochastic policies. They are guaranteed to converge to a locally optimal function even when complete information about the state is unavailable. They can also directly be applied to domains where the actions available to the agent lie in a continuous space (like motor control in robotics). As the previous example demonstrates, the policy function is often simpler and requires fewer parameters than the value function. However, state of the art policy gradient algorithms use a fixed parametrization, with less work on how the policy representation could be learned or improved.

The primary contribution of this thesis is to present two simple algorithms which can learn an adaptive representation of policy using policy gradient. The first is the Policy Tree algorithm, described in detail in Chapter 4. It aims to directly learn a function representing the policy, avoiding representation of value. This function takes the form of a decision tree, where the decision nodes test single feature variables, and the leaves of the tree contain a parametrized representation of a base policy. When the base policy can be represented in terms of linear functions of features, the Policy Tree algorithm is shown to be equivalent to replacing the original features with conjunctions of features. This leads to a variation called the Policy Conjunction algorithm, described in Chapter 6, which directly introduces such conjunctions without replacing the original features. In both of these algorithms, the representation is grown only when doing so improves the expected cumulative reward earned by following the policy, and not to increase the prediction accuracy of a value function. These algorithms are validated on a set of domains inspired by arcade games.

Chapter 2

The Reinforcement Learning Problem

As described in the previous chapter, reinforcement learning is concerned with creating agents that can interact with an unknown environment, and learn to behave in a way such as to maximise the cumulative reward attained. In this chapter, this view of reinforcement learning is formalized. The notation and definitions largely follow that of Sutton and Barto (1998).

2.1 The Environment

The environment is formulated as a Markov Decision Process (MDP) (Puterman, 2009) with the following components:

- a set of *states* $\mathcal{S}$.
- a set of *actions* $\mathcal{A}$.
- an *environmental dynamics* function $P : \mathcal{S} \times \mathcal{A} \to \text{DIST}(\mathcal{S} \times \mathbb{R})$.

where $\text{DIST}(\mathcal{X})$ is the set of all probability distributions over the set $\mathcal{X}$.

At each time step $t \in \{0, 1, 2, ...\}$, the agent is in a state $S_t \in \mathcal{S}$ and selects an action $A_t \in \mathcal{A}$. It receives a real valued *reward* $R_{t+1} \in \mathbb{R}$ and transitions to the next state S_{t+1}, both of which are drawn from the probability distribution $P(S_{t+1}, R_{t+1}|S_t, A_t)$. If the agent reaches a special state called the *terminal state*, the agent-environment interaction terminates, and no further rewards or states are observed.

It is assumed that each state S_t can be *factored* into a D-dimensional binary feature vector $\boldsymbol{\phi}(S_t) \in \{0, 1\}^D$. The purpose of factoring is to allow generalization

in the learning agent. Two different feature vectors, which share many common elements, may correspond to similar states. There exist methods, such as Coarse Coding, Tile Coding or Kanerva Coding (Sutton and Barto, 1998), which can convert a set of real valued parameters (like position or velocity) associated with the state into such a binary feature vector.

2.2 The Agent

The agent implements a function mapping states to the probability of choosing actions. This is known as the *policy* function $\pi_\theta : \mathcal{S} \rightarrow \text{DIST}(\mathcal{A})$. The agent chooses an action a at each time step by drawing from the probability distribution $\pi_{\boldsymbol{\theta}}(\cdot|S_t)$, where $\boldsymbol{\theta}$ represents an internal parametrization of its policy.

2.3 The Goal

The set of all the actions, states and rewards observed over one episode of learning of length T is called a *trajectory*, denoted by $\tau = \{S_t, A_t, R_{t+1} \ \forall t \in \{0, ..., T-1\}\}$. The cumulative reward obtained over an episode is called the *return*, denoted by $R(\tau)$. A general formulation for the return is a weighted sum of all rewards obtained during an episode:

$$R(\tau) = \sum_{t=0}^{T-1} w_t R_{t+1}. \tag{2.1}$$

There are two kinds of tasks usually encountered in reinforcement learning:

1. **Episodic Tasks:** In these problems, the terminal state is always encountered, and so the episode length T is always finite. The learning agent interacts with the environment over multiple episodes of learning. The return is well defined for $w_t = 1$, often called the *total reward* formulation.

2. **Continuing Tasks:** In these problems, it is possible that $T \rightarrow \infty$, and the agent interacts with the environment in one single episode of learning. Hence, modifying the policy during the episode is essential for learning in such tasks. The return is unbounded for $w_t = 1$, and a usual choice is to choose $w_t = \gamma^t$, where $\gamma \in [0, 1)$ is called the *discount factor*.

The goal of a reinforcement learning agent is to find a policy which maximises the *expected return*, which is denoted as $\rho(\boldsymbol{\theta})$:

$$\begin{aligned} \rho(\boldsymbol{\theta}) &= \mathbb{E}[R(\tau)] \\ &= \int_{\tau} R(\tau) \Pr(\tau). \end{aligned} \tag{2.2}$$

Note that the probability distribution over τ depends on both the environmental dynamics function P, and the agents policy $\pi_{\boldsymbol{\theta}}$. As the only thing which can be controlled by the agent is its internal parameter $\boldsymbol{\theta}$, the expected return can be viewed as a function of these parameters, from the perspective of the agent. In the next chapter, a particular class of reinforcement learning algorithms is introduced, in which gradient optimization is used to find a value of $\boldsymbol{\theta}$ which is locally optimal with respect to the expected return.

Chapter 3

Policy Gradient

Policy gradient algorithms work by applying gradient ascent to find a policy which maximizes the expected return. Note that computing the gradient $\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta})$ would involve an expectation over observed rewards, with the underlying probability distribution being a function of both the policy and the model of the environment. The model is unknown to the agent, but a sample estimate of the gradient can be obtained by observing trajectories of observations and rewards, while acting on-policy. This is the principle which underlies all policy gradient algorithms. In this chapter, two such algorithms REINFORCE (Williams, 1992) and GPOMDP (Baxter and Bartlett, 2000), shall be derived. There are many other policy gradient algorithms based on the Policy Gradient Theorem (Sutton et al., 2000), like Natural Actor-Critic (Peters and Schaal, 2008a) which learn both a policy and a value function. This thesis shall not describe them in detail, although the algorithms presented in the future chapters are readily applicable to them. Finally in this chapter, specific parametrizations of policy that can be applied to factored state representations are presented.

3.1 The REINFORCE Algorithm

The REINFORCE algorithm (Williams, 1992) is a way to obtain a Monte Carlo estimate of the gradient $\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta})$. Given the environmental dynamics function P and the policy $\pi_{\boldsymbol{\theta}}$, one can calculate the probability of obtaining a trajectory τ:

$$\Pr(\tau) = \Pr(S_0) \prod_{t=0}^{T-1} \left(\pi_{\boldsymbol{\theta}}(A_t|S_t) P(S_{t+1}, R_{t+1}|S_t, A_t)\right), \tag{3.1}$$

where $\Pr(S_0)$ is the probability of obtaining S_0 as the starting state. Now, taking the logarithm:

$$\log \Pr(\tau) = \log \Pr(S_0) + \sum_{t=0}^{T-1} \log \pi_{\boldsymbol{\theta}}(A_t|S_t) + \sum_{t=0}^{T-1} \log P(S_{t+1}, R_{t+1}|S_t, A_t). \tag{3.2}$$

By taking the gradient with respect to $\boldsymbol{\theta}$, the terms of this expression which are dependent on the environment dynamics disappear, as they do not depend on $\boldsymbol{\theta}$:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \log \Pr(\tau) &= \sum_{t=0}^{T-1} \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}}(A_t|S_t)} \\ \frac{\nabla_{\boldsymbol{\theta}} \Pr(\tau)}{\Pr(\tau)} &= \sum_{t=0}^{T-1} \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}}(A_t|S_t)} \\ \nabla_{\boldsymbol{\theta}} \Pr(\tau) &= \Pr(\tau) \sum_{t=0}^{T-1} \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}}(A_t|S_t)}. \end{aligned} \tag{3.3}$$

If one takes the gradient of the expected return in Equation 2.2, and combines it with Equation 3.3, one can get:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \rho(\boldsymbol{\theta}) &= \int_{\tau} R(\tau) \nabla_{\boldsymbol{\theta}} \Pr(\tau) \\ &= \int_{\tau} R(\tau) \Pr(\tau) \sum_{t=0}^{T-1} \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}}(A_t|S_t)} \\ &= \int_{\tau} \Pr(\tau) F(\tau) \\ &= \mathbb{E}[F(\tau)], \end{aligned} \tag{3.4}$$

where $F(\tau) = \sum_{t=0}^{T-1} \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}}(A_t|S_t)} R(\tau)$. Computing the expected value of $F(\tau)$ requires knowledge of the environmental model P, which is unknown to the agent. However, a Monte Carlo estimate of this expectation can be obtained.

If the trajectories were generated by following the policy $\pi_{\boldsymbol{\theta}}$, they represent samples of the distribution $\Pr(\tau)$, and the sample average of $F(\tau)$ represents an unbiased estimate of the gradient. This will converge towards $\mathbb{E}[F(\tau)]$ as the number of sampled trajectories increases. And so:

$$\nabla_{\boldsymbol{\theta}} \rho(\boldsymbol{\theta}) \approx \widetilde{\nabla_{\boldsymbol{\theta}} \rho(\boldsymbol{\theta})} = \langle F(\tau) \rangle \,, \tag{3.5}$$

where $\langle X \rangle$ denotes the sample mean of random variable X. $F(\tau)$ is a random variable which can be computed from the observed trajectories and the policy parameters $\boldsymbol{\theta}$, and so its sample average can be computed by the agent. The procedure

Algorithm 1 The REINFORCE algorithm

$\widetilde{\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta})} \leftarrow 0$ ▷ The gradient estimate
$\boldsymbol{Z} \leftarrow 0$ ▷ An eligibility trace vector of the same size as $\boldsymbol{\theta}$
$R \leftarrow 0$ ▷ The return
while S_t is not the terminal state **do** ▷ S_t denotes the current state
    Choose action A_t according to $\pi_{\boldsymbol{\theta}}(\cdot|S_t)$
    Observe reward R_t
    $\boldsymbol{Z} \leftarrow \boldsymbol{Z} + \nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(A_t|S_t)/\pi_{\boldsymbol{\theta}}(A_t|S_t)$
    $R \leftarrow R + R_t$
end while

$\widetilde{\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta})} \leftarrow R\boldsymbol{Z}$

to calculate $\widetilde{\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta})}$ from the trajectory of a single episode, when $R(\tau)$ is defined as the total reward, is shown in Algorithm 1. The parameters can be updated by the gradient ascent rule:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\widetilde{\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta})}, \tag{3.6}$$

where α is a chosen step size. In practise, as long as the step size is sufficiently small, it is possible to use the noisy estimate from a single trajectory for each gradient step.

3.2 The Policy Gradient/GPOMDP Algorithm

The REINFORCE algorithm can obtain an unbiased estimate of the gradient, however this estimate usually has high variance. There exists a method to reduce this variance when the return is a sum of scalar rewards, as pointed out by Williams (1992). This technique was further described as the GPOMDP algorithm by Baxter and Bartlett (2000), who showed that the method is valid even when the environment is partially observable. The exact same estimate of the gradient can also be obtained from the Policy Gradient Theorem (Sutton et al., 2000). In some literature (Peters and Schaal, 2008b), this method is known as the Policy Gradient/GPOMDP algorithm, and that is the convention adopted here. In this section, this method is derived from REINFORCE by removing some of the terms in $F(\tau)$, which have an expected value of zero, but add to the variance of the estimator.

When $R(\tau)$ is defined as some weighted sum of the individual rewards, as in

Equation 2.1, one can write $\mathbb{E}[F(\tau)]$ in Equation 3.4 as follows:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta}) &= \mathbb{E}[F(\tau)] \\
&= \mathbb{E}\left[\left(\sum_{t=0}^{T-1}\frac{\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(A_t|S_t)}{\pi_{\boldsymbol{\theta}}(A_t|S_t)}\right)\left(\sum_{t=0}^{T-1} w_t R_{t+1}\right)\right] \\
&= \sum_{i=0}^{T-1}\sum_{j=0}^{T-1}\mathbb{E}\left[\frac{\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(A_i|S_i)}{\pi_{\boldsymbol{\theta}}(A_i|S_i)} w_j R_{j+1}\right] \\
&= \sum_{i=0}^{T-1}\sum_{j=0}^{T-1}\mathbb{E}\left[\int_{A_i\in\mathcal{A}}\frac{\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(A_i|S_i)}{\pi_{\boldsymbol{\theta}}(A_i|S_i)} w_j R_{j+1}\pi_{\boldsymbol{\theta}}(A_i|S_i)dA_i\right] \\
&= \sum_{i=0}^{T-1}\sum_{j=0}^{T-1}\mathbb{E}\left[\int_{A_i\in\mathcal{A}}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(A_i|S_i) w_j R_{j+1}dA_i\right].
\end{aligned} \tag{3.7}$$

Intuitively, one would expect that past rewards do not depend on future actions. That is, for any $j < i$, R_{j+1} can be shown to be independent of A_i (Baxter and Bartlett, 2000). For any such i and j:

$$\mathbb{E}\left[\int_{A_i\in\mathcal{A}}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(A_i|S_i) w_j R_{j+1}dA_i\right] = w_j\mathbb{E}\left[R_{j+1}\right]\mathbb{E}\left[\int_{A_i\in\mathcal{A}}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(A_i|S_i)dA_i\right]. \tag{3.8}$$

As $\pi_{\boldsymbol{\theta}}$ is a probability distribution:

$$\int_{a_i\in\mathcal{A}}\pi_{\boldsymbol{\theta}}(A_i|S_i)dA_i = 1, \tag{3.9}$$

therefore,

$$\int_{a_i\in\mathcal{A}}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(A_i|S_i)dA_i = 0. \tag{3.10}$$

This shows that Equation 3.8 evaluates to 0. From Equations 3.7, 3.8 and 3.10 the calculation of $\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta})$ can be simplified as follows:

$$\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta}) = \mathbb{E}\left[\sum_{i=0}^{T-1}\sum_{j=i}^{T-1}\frac{\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(A_i|S_i)}{\pi_{\boldsymbol{\theta}}(A_i|S_i)} w_j R_{j+1}\right]. \tag{3.11}$$

The Monte Carlo estimate of this integral can be calculated by the agent, and used to perform gradient ascent:

$$\widetilde{\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta})} = \left\langle\sum_{i=0}^{T-1}\sum_{j=i}^{T-1}\frac{\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(A_i|S_i)}{\pi_{\boldsymbol{\theta}}(A_i|S_i)} w_j R_{j+1}\right\rangle. \tag{3.12}$$

The procedure to calculate this estimate from the trajectory of a single episode under the total reward formulation is presented in Algorithm 2. This method can also be used for continuing tasks (Baxter and Bartlett, 2001).

Algorithm 2 The Policy Gradient/GPOMDP algorithm

$\widetilde{\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta})} \leftarrow 0$ ▷ The gradient estimate
$\boldsymbol{Z} \leftarrow 0$ ▷ An eligibility trace vector with the same size as $\boldsymbol{\theta}$
while S_t is not the terminal state **do** ▷ S_t denotes the current state
  Choose action A_t according to $\pi_{\boldsymbol{\theta}}(\cdot|S_t)$
  Observe reward R_t
  $\boldsymbol{Z} \leftarrow \boldsymbol{Z} + \nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(A_t|S_t)/\pi_{\boldsymbol{\theta}}(A_t|S_t)$
  $\widetilde{\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta})} \leftarrow \widetilde{\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta})} + R_t\boldsymbol{Z}$
end while

3.3 Using Baselines to Reduce Variance

If one were to subtract a constant baseline b_{t+1} from each reward R_{t+1} in the trajectory, $\mathbb{E}[F(\tau)]$ in Equation 3.7 becomes:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta}) &= \sum_{i=0}^{T-1}\sum_{j=0}^{T-1}\mathbb{E}\left[\int_{A_i\in\mathcal{A}}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(A_i|S_i)w_j(R_{j+1}-b_{j+1})dA_i\right]\\ &= \sum_{i=0}^{T-1}\sum_{j=i}^{T}\mathbb{E}\left[\int_{A_i\in\mathcal{A}}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(A_i|S_i)w_jR_{j+1}dA_i - \int_{A_i\in\mathcal{A}}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(A_i|S_i)w_jb_{j+1}dA_i\right]\\ &= \sum_{i=0}^{T-1}\sum_{j=0}^{T-1}\mathbb{E}\left[\int_{A_i\in\mathcal{A}}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(A_i|S_i)w_jR_{j+1}dA_i\right].\end{aligned} \tag{3.13}$$

Therefore, the addition of a constant baseline does not introduce bias in the gradient estimator, although it can affect the variance. Greensmith et al. (2004) suggest an optimal baseline corresponding to minimum variance, in which b_t is a weighted average of rewards obtained at time step t.

3.4 Policy Parametrizations for Factored State

All of the algorithms described in this chapter can be applied to any policy $\pi_{\boldsymbol{\theta}}$ that is differentiable with respect to all of its parameters. In other words, $\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(\cdot|S_t)$ should exist. Here, some common ways to define such a policy function are discussed.

3.4.1 Linear Gibbs Softmax Policy

As discussed in Section 2.1, the assumption here is that each state s is associated with a D-dimensional binary feature vector $\boldsymbol{\phi}$. This parametrization also assumes

that the set of actions is finite, and associates $|\mathcal{A}|$ D-dimensional real valued parameter vectors $\boldsymbol{\theta_a}$ corresponding to each action $a \in \mathcal{A}$. The linear Gibbs softmax policy associates with each action a linear function of the feature vector, the magnitude of which represents the desirability of choosing that action. A softmax function is applied over the magnitudes for each action to generate a probability distribution:

$$\begin{aligned} \pi_{\boldsymbol{\theta}}(a|S_t) &= \frac{H(S_t, a)}{\sum_{i \in \mathcal{A}} H(S_t, i)}, \\ &\text{where,} \\ H(S_t, a) &= \exp(\boldsymbol{\theta_a}^T \boldsymbol{\phi}(S_t)), \end{aligned} \tag{3.14}$$

3.4.2 Normally Distributed Policy

When $\mathcal{A}$ is a space of real numbers, a policy suggested by Williams (1992) consists of a normal (or Gaussian) distribution, the mean and standard deviation of which are dependent on the state:

$$\begin{aligned} \pi_{\boldsymbol{\theta}}(a|S_t) &= \mathcal{N}(a|\mu(S_t), \sigma(S_t)), \\ &\text{where,} \\ \mu(S_t) &= \boldsymbol{\theta_\mu}^T \boldsymbol{\phi}(S_t), \\ \sigma(S_t) &= \exp(\boldsymbol{\theta_\sigma}^T \boldsymbol{\phi}(S_t)), \\ \mathcal{N}(a|\mu, \sigma) &= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(a-\mu)^2}{2\sigma^2}\right). \end{aligned} \tag{3.15}$$

3.4.3 Multi-Armed Bandit Policy

Another possible parametrization consists of a single scalar parameter θ_a per action, over which a softmax function is applied. This corresponds to a multi-armed bandit agent, which has a fixed probability of taking each action, independent of the state:

$$\pi_{\boldsymbol{\theta}}(a|S_t) = \frac{\exp(\theta_a)}{\sum_{i=1}^{|\mathcal{A}|} \exp(\theta_i)}. \tag{3.16}$$

3.5 Non-Parametric Policy Gradients

One notable work on adaptive (or non-parametric) representation for policy gradient includes the NPPG algorithm (Kersting and Driessens, 2008). In each iteration of the algorithm, a regression model is learned over a batch of data, with the functional gradient as its target. The final policy is a weighted sum of these models.

The regression model can be any complex function of the data, including decision trees. A disadvantage of this method is that each gradient step adds a new model to the policy, increasing the computational cost of action selection, and degrading the generalization ability (Da et al., 2014). Additionally, the functional gradient, as the derivative of the value, could be as complex as the value function itself. And so, as with value-based representation learning, a more complex representation may be learned than is necessary.

In the next chapter, an algorithm which can adapt its representation of policy, yet does not attempt to model the value of the gradient, shall be described.

Chapter 4

The Policy Tree Algorithm

This chapter describes the Policy Tree algorithm, which consists of a base parametric representation of policy and a binary decision tree. The tree divides the state space into distinct regions corresponding to its leaf nodes, in each of which a separate instantiation of the base policy is learned.

A decision tree architecture is commonly used in supervised learning, where the problem is to find an accurate function mapping the input space to a label. In such problems, the decision tree maps each input to a leaf node, and the output of the function is usually the sample average of the training examples mapped to the same leaf node. There are many standard algorithms to construct a tree in such a case, like C4.5 (Quinlan, 1993), in which the aim is to maximize the mutual information of the label and the leaf node.

The Policy Tree does not deal with labelled examples, and the standard algorithms used for classification and regression trees cannot be applied to learn the decision tree structure. Instead, the criterion used to grow the tree is to find the split which corresponds to the maximum increase in the expected return in a local region of the parameter space. The policy gradient algorithms described in the previous chapter are used to measure this criterion.

4.1 Notation

The internal nodes of the tree are decisions on an element of the feature vector. The index of this element is called the *decision index* of the node. An internal node with decision index i maps a state S_t to one of its two child nodes, based on the value of $\phi^{(i)}(S_t)$ (the ith element of the vector $\boldsymbol{\phi}(S_t)$). Every state S_t maps to one leaf node $l(S_t)$ in the tree, which is associated with a real valued parameter vector $\boldsymbol{\theta}_{l(\boldsymbol{S_t})}$. The

base policy at the leaf is parametrized by this vector and is denoted by $\pi_{\boldsymbol{\theta}_{l(\boldsymbol{S_t})}}(\cdot|S_t)$. This could be the policy functions in Equations 3.14, 3.15, 3.16 or any other valid parametrization of policy.

4.2 Overview of the Policy Tree Algorithm

The high level procedure can be described as:

1. Start with a single-node decision tree, with its root node containing a randomly initialized parametrization of the base policy.
2. Optimize all leaf node parameters using policy gradient for a fixed number of episodes or time steps.
3. Keep the parameters fixed for a number of episodes or time steps, while the merit of each split is judged. Choose a leaf node and an observation index to split, according to our tree growth criterion. Create two new children of this node, which inherit the same policy parameters as their parent. Go to step 2 and repeat.

The steps 2 and 3 of the algorithm are described in detail in the following sections.

4.3 Parameter Optimization

During this phase, the tree structure is kept fixed, while the parameters are optimized using a policy gradient algorithm, such as Algorithm 1 or 2. The per-step computational complexity during this phase depends on the actual algorithm and parametrization used. For most policy gradient algorithm, this would be linear in the number of parameters, which is $\mathcal{O}(N_L N_P)$, where N_L is the number of leaf nodes and N_P is the number of parameters in the base policy.

4.4 Tree Growth

In this phase, the structure of the tree is altered by splitting one of the leaf nodes, changing the underlying representation. In order to choose a good candidate split,

one would ideally like to know the global effect of a split after optimizing the resulting tree. This would require making every candidate split and performing parameter optimization in each case, which is unrealistic and inefficient. However, if one were to suggest a candidate split, and keep its parameters fixed, the gradient of the expected return of this new policy function provides a first order approximation of the expected return. This approximation is valid within a small region of the parameter space, and can be used to measure the local effect of the split. This is the basis of our criterion to grow the tree. First, a method to calculate the gradients corresponding to every possible split in the tree is described.

A valid addition to the policy tree involves a split on one of the leaf nodes, on a parameter $k \in \{0, ..., D-1\}$, such that k is not a decision index on the path from the leaf node to the root. For every leaf node L in the tree, and for every valid index k in $\{0, ..., D-1\}$, a pair of *fringe* child nodes are created, denoted by $F_{L,k}$ and $F'_{L,k}$. They represent the child nodes of L which would be active when $\phi^{(k)} = 1$ and $\phi^{(k)} = 0$, respectively. Both of these nodes are associated with a parameter vector which is the same as that of the parent leaf node, that is, for all L and k:

$$\boldsymbol{\theta}_{\boldsymbol{F_{L,k}}} = \boldsymbol{\theta}_{\boldsymbol{F'_{L,k}}} = \boldsymbol{\theta_L}. \tag{4.1}$$

Let $\boldsymbol{\psi_{L,k}}$ denote the combined vector of all the parameters associated with the tree, when it is expanded to include the pair of fringe nodes $F_{L,k}$ and $F'_{L,k}$. This vector is a concatenation of the vectors $\boldsymbol{\theta}_{\boldsymbol{F_{L,k}}}$, $\boldsymbol{\theta}_{\boldsymbol{F'_{L,k}}}$ and $\boldsymbol{\theta_{L'}}$ for all leaf nodes $L' \neq L$. Note that each such vector corresponds to a different policy function, which is denoted by $\pi_{\boldsymbol{\psi_{L,k}}}$. Let $\rho(\boldsymbol{\psi_{L,k}})$ denote the corresponding expected return.

Equation 4.1 ensures that $\pi_{\boldsymbol{\psi_{l(S_t),k}}}(\cdot|S_t) = \pi_{\boldsymbol{\theta_{l(S_t)}}}(\cdot|S_t)$, which means that all these policies have the same distribution over actions as the one represented by the existing policy tree, even though the underlying representation has changed. This ensures that a correct sample estimate of the fringe gradient $\nabla_{\boldsymbol{\psi_{L,k}}}\rho(\boldsymbol{\psi_{L,k}})$ can be measured by following the policy represented by the tree. It is important to obtain a good estimate of these gradients to avoid making spurious splits based on noise. Therefore, during this phase, the policy is kept fixed while a suitably large number of trajectories are observed, and $\nabla_{\boldsymbol{\psi_{L,k}}}\rho(\boldsymbol{\psi_{L,k}})$ is estimated for all L and k.

The leaf node L to split on, and the split index k, is chosen by the following procedure:

$$L, k = \arg\max_{L,k} \; ||\nabla_{\boldsymbol{\psi_{L,k}}}\rho(\boldsymbol{\psi_{L,k}})||_q. \tag{4.2}$$

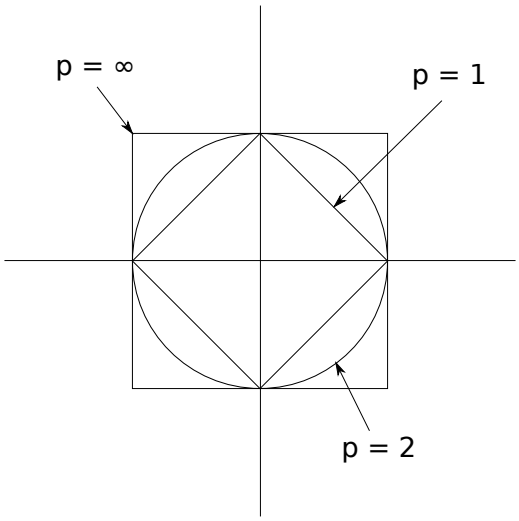

Figure 4.1: The p-norm spheres representing $||\boldsymbol{\Delta\theta}||_p = \epsilon$

It is worth reflecting on the interpretation of the q-norm of the gradient vector, in order to understand the above criterion. By using a first order Taylor expansion of the expected return, one can measure the change corresponding to a tiny step $\boldsymbol{\Delta\psi}$:

$$\rho(\boldsymbol{\psi_{L,k}} + \boldsymbol{\Delta\psi}) = \rho(\boldsymbol{\psi_{L,k}}) + \nabla_{\boldsymbol{\psi_{L,k}}}\rho(\boldsymbol{\psi_{L,k}})^T\boldsymbol{\Delta\psi}. \tag{4.3}$$

If $\boldsymbol{\Delta\psi}$ is constrained to lie within a small p-norm sphere with radius ϵ, then:

$$\begin{gathered}\max_{\{\boldsymbol{\Delta\psi}\,:\,||\boldsymbol{\Delta\psi}||_p \leq \epsilon\}} \nabla_{\boldsymbol{\psi_{L,k}}}\rho(\boldsymbol{\psi_{L,k}})^T\boldsymbol{\Delta\psi} = ||\nabla_{\boldsymbol{\psi_{L,k}}}\rho(\boldsymbol{\psi_{L,k}})||_q\epsilon, \\ \text{where, } \frac{1}{p} + \frac{1}{q} = 1.\end{gathered} \tag{4.4}$$

This shows that the q-norm of the gradient represents the maximum change in the objective function within a local region of the parameter space bounded by the p-norm sphere, where p and q are the dual norms of each other (Kolmogorov and Fomin, 1957). Figure 4.1 shows a graphical representation of various p-norm sphere for two dimensions.

By the same reasoning, $||\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta})||_q$ represents the maximum local improvement in the expected return that can be obtained without altering the representation of the tree. A simple stopping condition for the tree expansion is $||\nabla_{\boldsymbol{\psi_{L,k}}}\rho(\boldsymbol{\psi_{L,k}})||_q < \lambda||\nabla_{\boldsymbol{\theta}}\rho(\boldsymbol{\theta})||_q$, for some λ.

The fringe gradient $\nabla_{\boldsymbol{\psi_{L,k}}}\rho(\boldsymbol{\psi_{L,k}})$ has $(N_L + 1)N_P$ components, as it is defined over all the parameters corresponding to an incremented tree. However, $(N_L - 1)N_P$ of these components are partial derivatives with respect to the existing

parameters in the tree, and are shared by all the fringe gradients. Thus $N_L D$ gradients involving $2N_P$ unique parameters need to be measured, and the per-step computational complexity during this phase when using REINFORCE or GPOMDP is $\mathcal{O}(N_P N_L D)$. Note that the length of this phase will almost always be considerably lower than the previous one, as making an accurate gradient estimate is simpler than optimizing the parameters.

4.4.1 Fringe Bias Approximation

For most base policies, the number of parameters will increase with the number of features, making the complexity of the tree growth phase quadratic (or worse) in the number of features. Here, an approximation which can reduce this complexity is described, when the base policy depends on linear functions of the features. If there are N_F such functions, then $N_P = N_F D$. An example of this is the linear Gibbs softmax policy (Equation 3.14), where $N_F = |\mathcal{A}|$.

The standard practise when defining a linear function is to augment the input vector with a bias term, usually chosen as the first term of the vector. This term, denoted as $\phi^{(0)}(S_t)$, is always 1. If one were to choose a few components to represent the gradient of the fringe parameters, choosing the parameters associated with $\phi^{(0)}(S_t)$ is a reasonable choice. Let $\boldsymbol{\theta}^{\mathbf{0}}_{\boldsymbol{F_{L,k}}}$ represent the vector of N_F parameters that are associated with this feature in fringe node $F_{L,k}$.

To apply this approximation to the Gibbs policy in Equation 3.14 as the base, $\nabla_{\boldsymbol{\theta}^{\mathbf{0}}_{\boldsymbol{F_{L,k}}}} \rho(\boldsymbol{\psi_{L,k}})$ is computed, and the other terms of the gradient are set to zero. The tree growth criterion remains the same, which is to measure the norm of the gradient in this reduced space. The computational complexity becomes $\mathcal{O}(N_L N_F D)$, which is the same as that for parameter optimization.

Chapter 5

Experiments on Policy Tree

In this section, the following questions are evaluated empirically:

1. Can the Policy Tree improve over the base policy?
2. How well does the fringe bias approximation work?
3. Is the improvement merely due to an increase in the number of parameters, or does Policy Tree choose intelligent splits?

To answer these questions, I implemented a set of domains inspired by arcade games.

5.1 Domains

The test suite is a set of 4 simple games, which have a 16x16 pixel game screen with 4 colours. A pictorial representation of them is presented in Figures 5.1 to 5.4. All of these games are episodic with a maximum episode length of 256 time steps, and every object moves with a speed of one pixel per step. Objects in these games, including the player agent, enemy agents, friendly agents or bullets are a single pixel in size, and each object type is of a distinct colour. Unless specified otherwise, the actions available to the agent are to move up, down, left, right or stay still.

These games contain elements of partial observability and non-linearity in the optimal policy function. As examples, the direction of objects in the games cannot be determined from a single game screen, and the best action to take is often conditional on multiple variables.

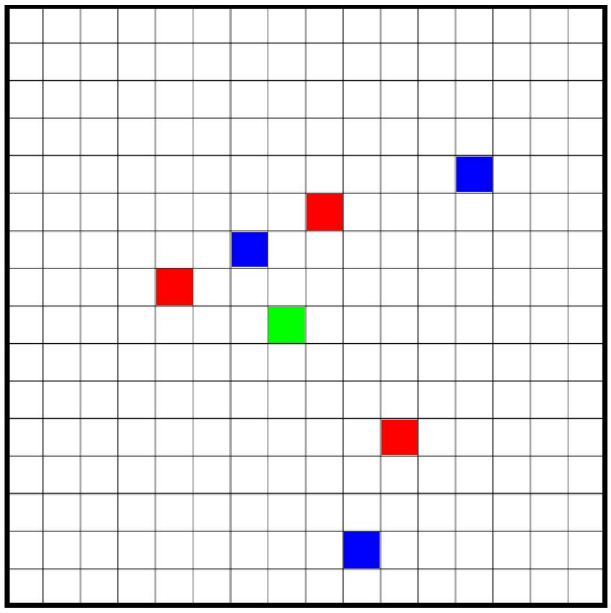

Figure 5.1: The graphical representation of Monsters

5.1.1 Monsters

The player agent is green. There are three red monsters and three blue power suits randomly located in the game area. The agent can collect and wear a power suit, allowing it to kill the next monster it collides with for one point. A power suit lasts a maximum of 16 time steps once it is worn. The monsters chase the agent when it is not powered, and stay still otherwise.

5.1.2 Switcheroo

There are two types of objects, red and blue. At regular intervals, groups of 8 objects of a randomly chosen type move across the screen horizontally. The player agent starts in the center of the screen, as an object of a random type. Hitting objects of the other type terminates the episode, while hitting an object of the same type earns a point. After a point is earned, the agent switches to the other type.

5.1.3 Mothership

The blue mothership sits at the top of the screen, while 4 rows of 8 red guard ships patrol horizontally below. The green player agent is constrained to the bottom of the screen, and cannot move up or down. It can shoot a bullet which moves one pixel upwards per time step. It needs to shoot the mothership to collect 5

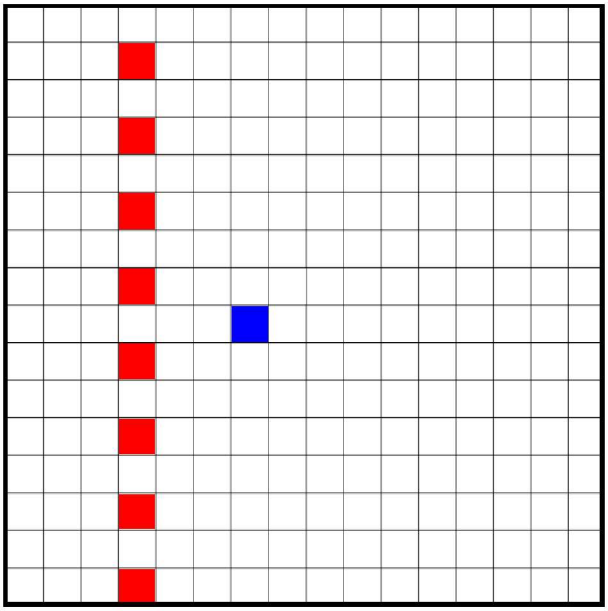

Figure 5.2: The graphical representation of Switcheroo

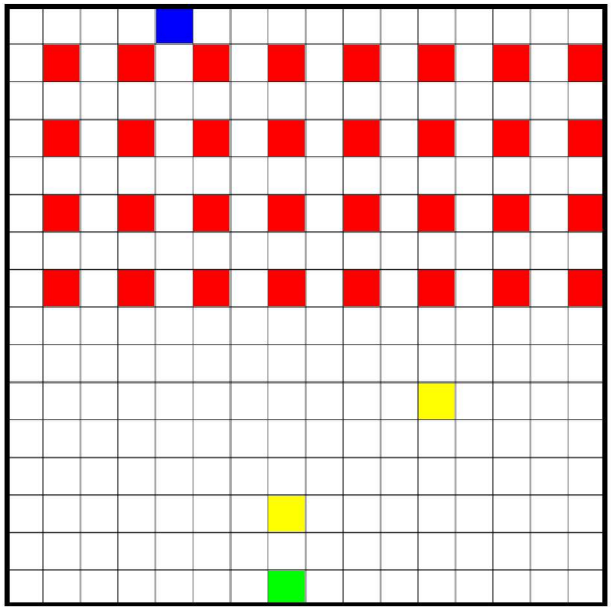

Figure 5.3: The graphical representation of Mothership

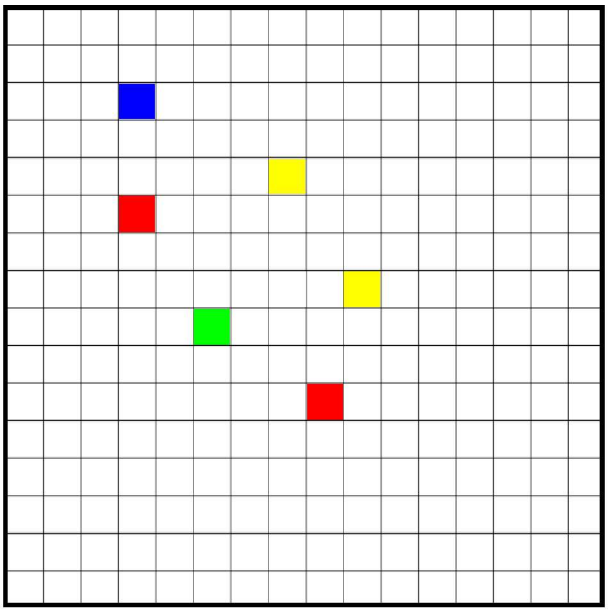

Figure 5.4: The graphical representation of Rescue

points, and then shoot the guard ships for 1 point each. Shooting the guards before destroying the mothership simply reflects bullets back at the agent, and alerts the enemy, causing the mothership to move randomly, and the guard ships to shoot bullets. The agent is only allowed to have one active bullet at a time, and there is a random wait of 0 or 1 time steps after its bullet is consumed. All of the bullets are coloured yellow.

5.1.4 Rescue

The green player agent has to collect yellow hostages and bring them to the blue rescue location to collect a point. The rescue location is randomly located after every successful rescue, and the player starts from a random location at the top of the screen. The hostages and red enemies move across the screen horizontally at random intervals, and collision with the enemies results in instant episode termination.

5.2 Feature Generation

For generating the binary feature vector, the location of the player agent is assumed to be known. The game screen is divided into zones, the size of which grows exponentially with the distance from the agent. Formally, if (x, y) represents the dis-

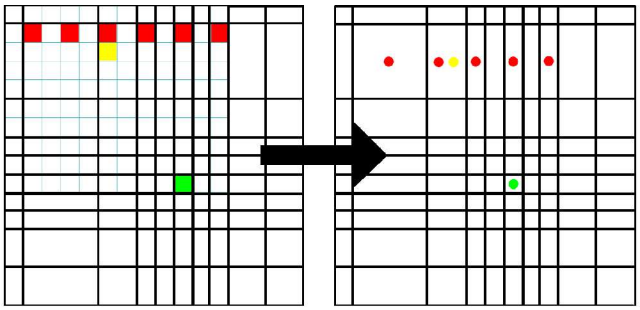

Figure 5.5: The feature mapping process.

placement of a pixel from the agent, all pixels with the same value of $\lfloor \log_2 |x| \rfloor$, $\lfloor \log_2 |x| \rfloor$, $\text{sgn}(x)$ and $\text{sgn}(y)$ belong to the same zone. For each zone, there is a feature corresponding to every colour used in the game. A feature is set to 1 if an object of the corresponding colour is found in that zone. An example of feature generation is shown in Figure 5.5. The left image shows a game screen for Mothership, with the black lines representing the zones centred around the agent. The right screen indicates the active features in each zone via coloured dots. 7 features are active in this example. Such a choice of features allows fine distinctions to be made between objects near the agent, while ensuring that the number of feature grows only logarithmically with the size of the game screen.

5.3 Base Policy

The linear Gibbs softmax policy is used as the base policy for the experiments, augmented with ϵ-greedy exploration. This policy is defined as:

$$\pi_{\boldsymbol{\theta_L}}(a|S_t) = (1-\epsilon)\frac{\exp(\boldsymbol{\theta_{L,a}}^T \boldsymbol{\phi}(S_t))}{\sum_{i=1}^{|\mathcal{A}|} \exp(\boldsymbol{\theta_{L,i}}^T \boldsymbol{\phi}(S_t))} + \frac{\epsilon}{|\mathcal{A}|}. \tag{5.1}$$

The ϵ term ensures that the policy is sufficiently stochastic even if θ takes on heavily deterministic values. This allows accurate estimation of the gradient at all times, and ensures that the policy does not quickly converge to a poor deterministic policy due to noise. Policy Tree benefits a great deal from exploration, as often highly optimized parameters are inherited from parent nodes, and need to be re-optimized after a split. The base policy (without the tree) was tested both with and without the ϵ term, and results show that its presence increases average performance.

5.4 Experimental Setup

Four different algorithms were tested on the domains. The first was the standard Policy Tree algorithm with the ϵ-greedy Gibbs policy as the base. The second was a version with the fringe bias approximation enabled. The third was a version of the algorithm which chooses a random split during tree growth, for the purpose of testing whether the Policy Tree just benefits from having a larger number of parameters, or whether it makes good representational choices. And finally, the base policy was tested, representing the standard parametric approach to policy gradient.

For Policy Tree, a parameter optimization stage of 49000 episodes and a gradient averaging phase during tree growth of 1000 episodes was used. Splits were therefore made after every 50000 episodes. The value of ϵ used was 0.001. For the tree growth criterion, the $q = 1$ norm of the gradient was chosen. Policy Gradient/GPOMDP (Algorithm 2) under the total reward formulation is used to measure the gradient in all of the algorithms. Additionally, the optimal baseline for reducing variance (Greensmith et al., 2004) is used. The stopping condition parameter λ was set to 1. A good step size during gradient optimization for the base policy was obtained via a parameter sweep for each domain (over the set $\alpha \in \{0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001\}$. The same step size was used for all the algorithms on that domain. For each algorithm, 30 different runs of learning over 500000 episodes were performed. The average return was measured as the moving average of the total reward per episode with a window length of 50000.

5.5 Results

The learning curves of the Policy Tree algorithm as compared to the base policy are shown in Figures 5.6 to 5.9. The standard error in the results, across the 30 runs, is represented by the vertical error bars in the graphs. These results allow us to answer the questions posed earlier:

1. The Policy Tree algorithm improves upon the underlying base policy with statistical significance.

2. The fringe bias approximation does not do as well as the exact measure in most domains, but still does improve over the linear parametrization in all 4

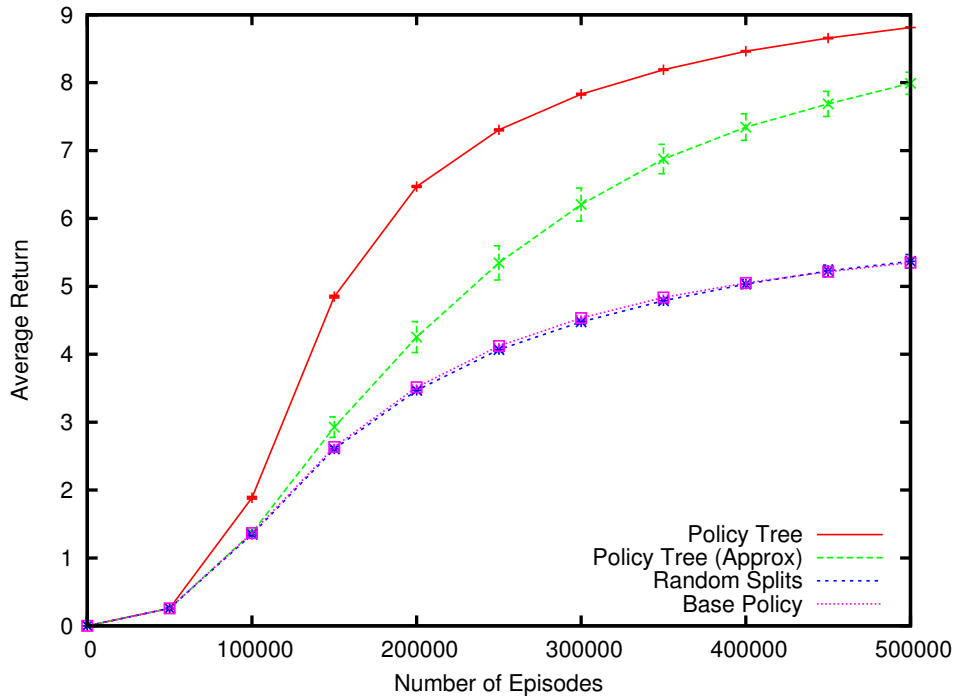

Figure 5.6: Results of the Policy Tree algorithm on Monsters

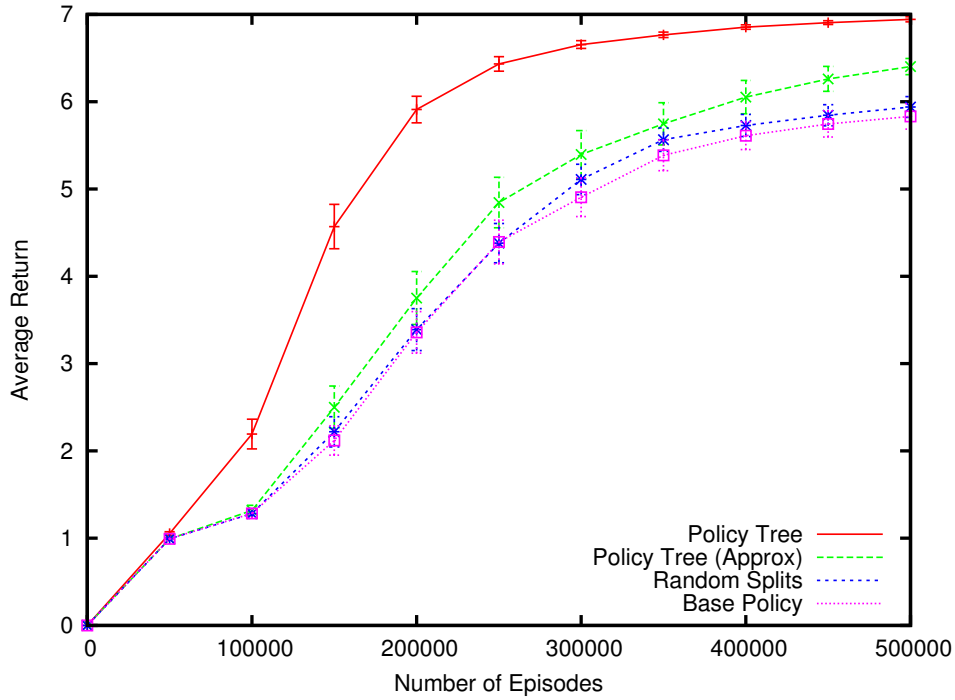

Figure 5.7: Results of the Policy Tree algorithm on Switcheroo

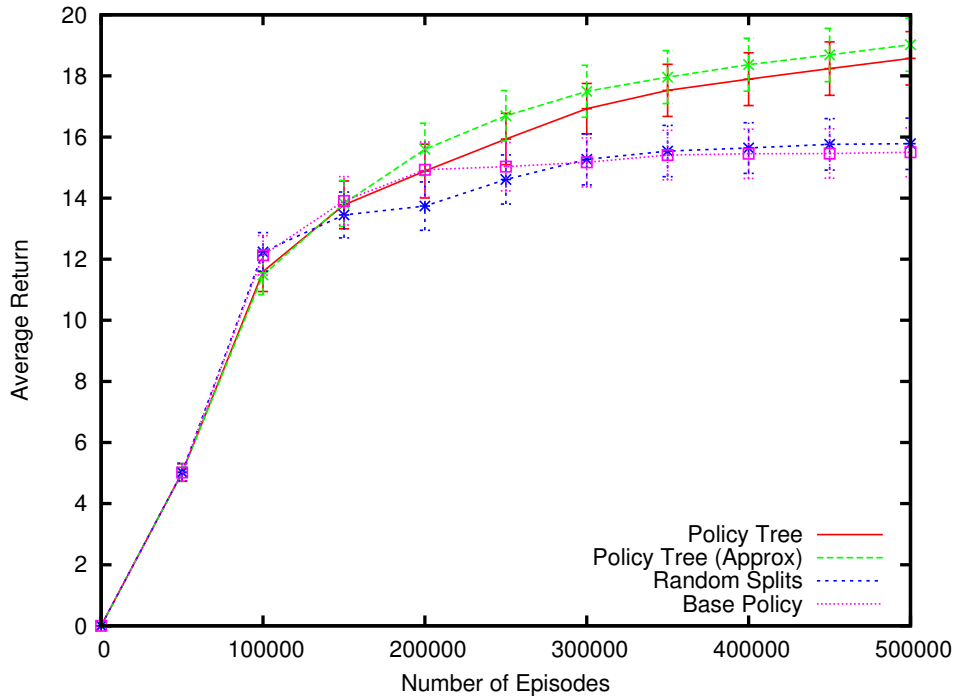

Figure 5.8: Results of the Policy Tree algorithm on Mothership

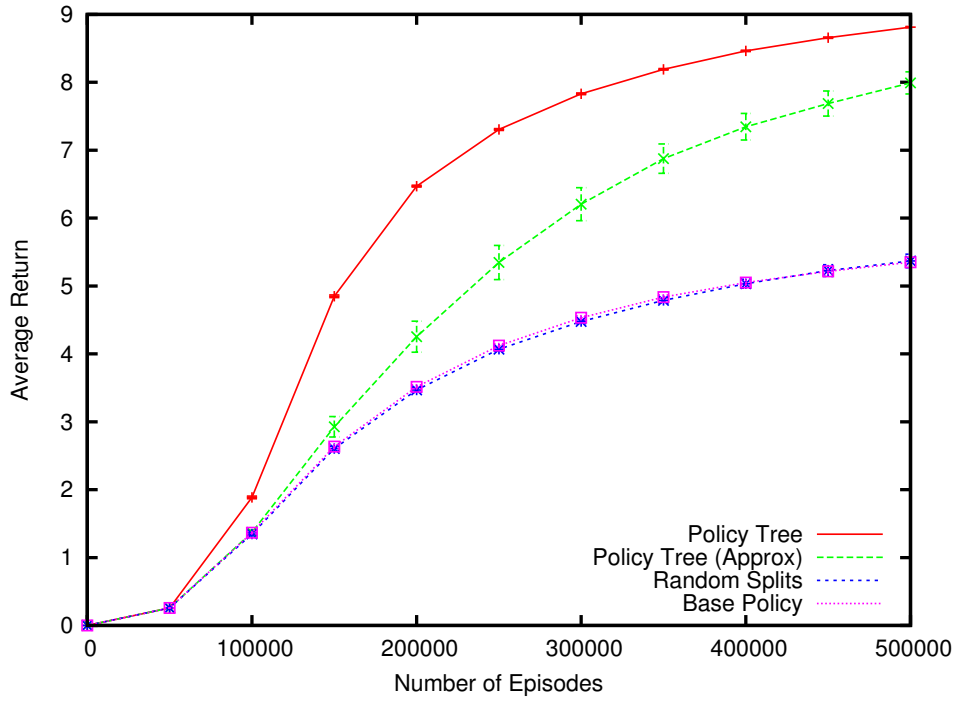

Figure 5.9: Results of the Policy Tree algorithm on Rescue

games significantly, without enduring additional computational complexity during the tree growth phase.

3. An arbitrary increase in the number of parameters via the random splitting does not improve performance at all. This shows that the tree growth criterion contributes significantly to the effectiveness of the algorithm.

The Monsters domain shows the biggest improvement, as well as the greatest gap between the exact and approximate versions. In this domain, the feature that represents whether or not the agent is powered is very informative. Policy Tree chooses this as the first split in 80% of the runs, while this drops to 20% with the approximation enabled. However, the approximate version was found to outperform the base policy even when this split was not made over the course of learning, indicating that the chosen splits are not meaningless.

Chapter 6

The Policy Conjunction Algorithm

The Policy Tree algorithm described in the previous chapter is applicable to any differentiable base policy function. Some commonly used policy functions depend on the state only through linear functions of the feature vector. These policies are called *semi-linear policies* in this chapter for convenience. The first section of this chapter shows that for such policies the Policy Tree function can be thought of as replacing existing features in the feature vector with higher-order conjunctions. The next sections develop a variation on Policy Tree called the Policy Conjunction algorithm, which is restricted to working with semi-linear policies. It can adapt the representation by including an arbitrary number of higher order feature conjunctions during the growth stage, and does not remove the original features. As with Policy Tree, the representation change is chosen to maximize the local increase in expected return.

6.1 A Flat View of the Policy Tree Algorithm

If the Policy Tree algorithm is used with a semi-linear policy like the the normally distributed policy (Equation 3.15) as the base, the effective policy function at a leaf node L can be written as:

$$\pi_{\boldsymbol{\theta}_L}(\cdot \mid S_t) = \mathcal{N}\left(\cdot \mid \boldsymbol{\theta_{L,\mu}}^T \boldsymbol{\phi}(S_t), \exp\left(\boldsymbol{\theta_{L,\sigma}}^T \boldsymbol{\phi}(S_t)\right)\right), \tag{6.1}$$

The policy $\pi_{\boldsymbol{\theta}_L}(\cdot|S_t)$ is used to select actions if and only if L is the active leaf node, which is true when the conjunction of features corresponding to the decisions taken on the path from the root to the leaf node (denoted by $\mathcal{F}_L(S_t)$) is true. So, the

effective policy of the entire tree can be written as:

$$\pi_{\boldsymbol{\theta}}\left(\cdot \mid S_t\right) = \mathcal{N}\left(\cdot \mid \sum_L {\boldsymbol{\theta}_{\boldsymbol{L},\boldsymbol{\mu}}}^T \boldsymbol{\phi}\left(S_t\right) \mathcal{F}_L\left(S_t\right), \sum_L {\boldsymbol{\theta}_{\boldsymbol{L},\boldsymbol{\sigma}}}^T \boldsymbol{\phi}\left(S_t\right) \mathcal{F}_L\left(S_t\right)\right), \tag{6.2}$$

where the summation is over all the leaf nodes in the tree. Note that $\mathcal{F}_L\left(S_t\right)$ is non-zero only for the active leaf node. This can be simplified as:

$$\pi_{\boldsymbol{\theta}}(\cdot \mid S_t) = \mathcal{N}(\cdot \mid {\boldsymbol{\theta}_{\boldsymbol{\mu}}}^T \boldsymbol{\Phi}(S_t), {\boldsymbol{\theta}_{\boldsymbol{\sigma}}}^T \boldsymbol{\Phi}(S_t)), \tag{6.3}$$

where $\boldsymbol{\theta}_{\boldsymbol{\mu}}$, $\boldsymbol{\theta}_{\boldsymbol{\sigma}}$ and $\boldsymbol{\Phi}(S_t)$ are concatenations of $\boldsymbol{\theta}_{\boldsymbol{L},\boldsymbol{\mu}}$, $\boldsymbol{\theta}_{\boldsymbol{L},\boldsymbol{\sigma}}$ and $\boldsymbol{\phi}\left(S_t\right) \mathcal{F}_L\left(S_t\right)$, respectively, over all leaf nodes L. The effective policy of the tree thus uses the same probability function as that used in the base policy, although it uses an expanded feature vector $\boldsymbol{\Phi}(S_t)$.

The expanded features corresponding to leaf node L are of the form $\boldsymbol{\phi}^{(i)} \wedge \mathcal{F}_L$ for all $i \in \{0, ..., D-1\}$. A split on index k replaces these features with $\boldsymbol{\phi}^{(k)} \wedge \boldsymbol{\phi}^{(i)} \wedge \mathcal{F}_L$ and $\neg\boldsymbol{\phi}^{(k)} \wedge \boldsymbol{\phi}^{(i)} \wedge \mathcal{F}_L$, and is chosen when doing so leads to the best collective local improvement as measured by Equation 4.2. So, for semi-linear base policies, the Policy Tree algorithm can be seen as a way to perform adaptive feature expansion, with the following restrictions:

1. A feature corresponding to a leaf node cannot be individually expanded. It needs to be chosen for expansion along with all the other features corresponding to the same node.

2. The existing features corresponding to a leaf node are removed, once the node is split.

The next section describes the Policy Conjunction algorithm, which can be viewed as a variation of the Policy Tree algorithm that does not obey the above restrictions, but is only applicable in the case of linear base policies .

6.2 Overview of the Policy Conjunction Algorithm

1. Start with a randomly initialized linear base policy $\pi_{\boldsymbol{\theta}}$ with a feature vector $\boldsymbol{\Phi}$, where $\boldsymbol{\Phi}$ is initially equal to the original feature vector $\boldsymbol{\phi}$. The parameter vector $\boldsymbol{\theta}$ represents the combined vector of all the parameters used in the policy.

2. Optimize $\boldsymbol{\theta}$ using a policy gradient algorithm like Algorithm 1 or 2 for a fixed number of episodes or time steps.

3. The growth phase: Keep $\boldsymbol{\theta}$ fixed for a number of episodes or time steps, while the scores of all valid candidate features are judged (explained further in Section 6.3). Add the top C candidates to $\boldsymbol{\Phi}$. Append the corresponding parameters to $\boldsymbol{\theta}$ and set them to 0. Modify the step size as necessary (explained in Section 6.4). Go to step 2 and repeat.

6.3 Adding candidate features

Unlike the Policy Tree algorithm, the Policy Conjunction algorithm considers feature expansion on each individual feature in $\boldsymbol{\Phi}$. To measure the benefit of such an expansion, it considers replacing a feature $\Phi^{(i)}$ with $\phi^{(j)} \wedge \Phi^{(i)}$ and $\neg\phi^{(j)} \wedge \Phi^{(i)}$ for some j, such that neither of these new features are already present in $\boldsymbol{\Phi}$. Each feature introduces as many parameters to the representation as there are linear functions in the policy. For example, the normally distributed policy in Equation 6.3 has two parameters $\theta_\mu^{(i)}$ and $\theta_\sigma^{(i)}$ for each feature $\Phi^{(i)}$.

Let $\boldsymbol{\theta_{i,j}}$ and $\boldsymbol{\theta'_{i,j}}$ denote vectors of parameters corresponding to the candidates feature $\phi^{(j)} \wedge \Phi^{(i)}$ and $\neg\phi^{(j)} \wedge \Phi^{(i)}$, respectively, while $\boldsymbol{\theta_i} \subset \boldsymbol{\theta}$ denotes a vector of parameters for the original feature $\Phi^{(i)}$. Let $\boldsymbol{\psi}_{i,j}$ denote the concatenation of $\boldsymbol{\theta_{i,j}}$, $\boldsymbol{\theta'_{i,j}}$ and $\boldsymbol{\theta_k}$ for all $k \neq i$, representing the resulting feature vector after replacing $\Phi^{(i)}$ with $\phi^{(j)} \wedge \Phi^{(i)}$ and $\neg\phi^{(j)} \wedge \Phi^{(i)}$.

Let the policy function when using the feature vector $\boldsymbol{\psi}_{i,j}$ be denoted by $\pi_{\boldsymbol{\psi}_{i,j}}$ and its expected return be denoted by $\rho\left(\boldsymbol{\psi}_{i,j}\right)$. If $\boldsymbol{\theta_{i,j}} = \boldsymbol{\theta'_{i,j}} = \boldsymbol{\theta_i}$, then $\pi_{\boldsymbol{\psi}_{i,j}} = \pi_{\boldsymbol{\theta}}$, and one can obtain the gradient $\nabla_{\boldsymbol{\psi}_{i,j}}\rho\left(\boldsymbol{\psi}_{i,j}\right)$, for all valid values of i and j, by following the policy $\pi_{\boldsymbol{\theta}}$. The index $\text{best}\,(i)$ corresponding to the best split on feature $\Phi^{(i)}$ is obtained as:

$$\text{best}\,(i) = \arg\max_j \; ||\nabla_{\boldsymbol{\psi}_{i,j}}\rho\left(\boldsymbol{\psi}_{i,j}\right)||_q. \tag{6.4}$$

As argued in Section 4.4, this corresponds to the maximum local improvement in the policy. Similarly to the stopping condition in Policy Tree, expansion on feature i could be stopped when $||\nabla_{\boldsymbol{\psi}_{i,\text{best}(i)}}\rho\left(\boldsymbol{\psi}_{i,j}\right)||_q < \lambda||\nabla_{\boldsymbol{\theta}}\rho\left(\boldsymbol{\theta}\right)||_q$.

Now, it is also possible to obtain the representational power of the features $\phi^{(j)} \wedge \Phi^{(i)}$ and $\neg\phi^{(j)} \wedge \Phi^{(i)}$, by including one of them in $\boldsymbol{\Phi}$ while retaining $\Phi^{(i)}$. These two representational choices are compared in Table 6.1. As $\boldsymbol{\theta_i}$, $\boldsymbol{\theta_{i,j}}$ and $\boldsymbol{\theta'_{i,j}}$ can span

$\Phi^{(i)}$	$\phi^{(j)}$	Active Parameters ($\phi^{(j)} \wedge \Phi^{(i)}$ and $\neg\phi^{(j)} \wedge \Phi^{(i)}$ added)	Active Parameters ($\Phi^{(i)}$ retained, $\phi^{(j)} \wedge \Phi^{(i)}$ added)
0	0	–	–
0	1	–	–
1	0	$\boldsymbol{\theta'_{i,j}}$	$\boldsymbol{\theta_i}$
1	1	$\boldsymbol{\theta_{i,j}}$	$\boldsymbol{\theta_i} + \boldsymbol{\theta_{i,j}}$

Table 6.1: Choice of representation during feature expansion

the entire range of real valued vectors, the space of policies that can be represented in both cases is the same. However, they differ in terms of the candidate features they consider in subsequent growth phases. Retaining the original feature allows the algorithm to choose simpler features like $\phi^{(k)} \wedge \Phi^{(i)}$ in the future, for some $k \neq j$, reducing the harm of choosing a poor split. This is the choice made in the Policy Conjunction algorithm, and is one of its major departures from the Policy Tree algorithm. Note that as $\boldsymbol{\theta_i}$ is retained, $\boldsymbol{\theta_{i,j}}$ must be initialized to 0 such that the policy is unchanged.

Under first order Taylor approximation conditions, the local benefit of a split on each feature $\Phi^{(i)}$ can be computed independently. Therefore, one can choose the best C candidate features of the form $\phi^{(\text{best}(i))} \wedge \Phi^{(i)}$, with $||\nabla_{\boldsymbol{\psi}_{i,\text{best}(i)}} \rho \left(\boldsymbol{\psi}_{i,\text{best}(i)} \right) ||_q$ being the score, and add them to $\boldsymbol{\Phi}$.

6.4 Modifying the Step Size during Gradient Optimization

In general, as the number of active features increases, each gradient step causes a greater change in the policy's distribution over actions, making gradient optimization potentially unstable as the feature vector $\boldsymbol{\theta}$ expands. To address this, a simple fix is to keep the step size inversely proportional to the number of features. This is not needed in the Policy Tree algorithm, because at any time step, only the features corresponding to a single leaf node can be active.

6.5 Experiments

The Policy Conjunction algorithm was evaluated on the domains described in Section 5.1, and a comparison with Policy Tree is presented in Figures 6.1 to 6.4. The same base policy described in Equation 5.1 was used for both algorithms.

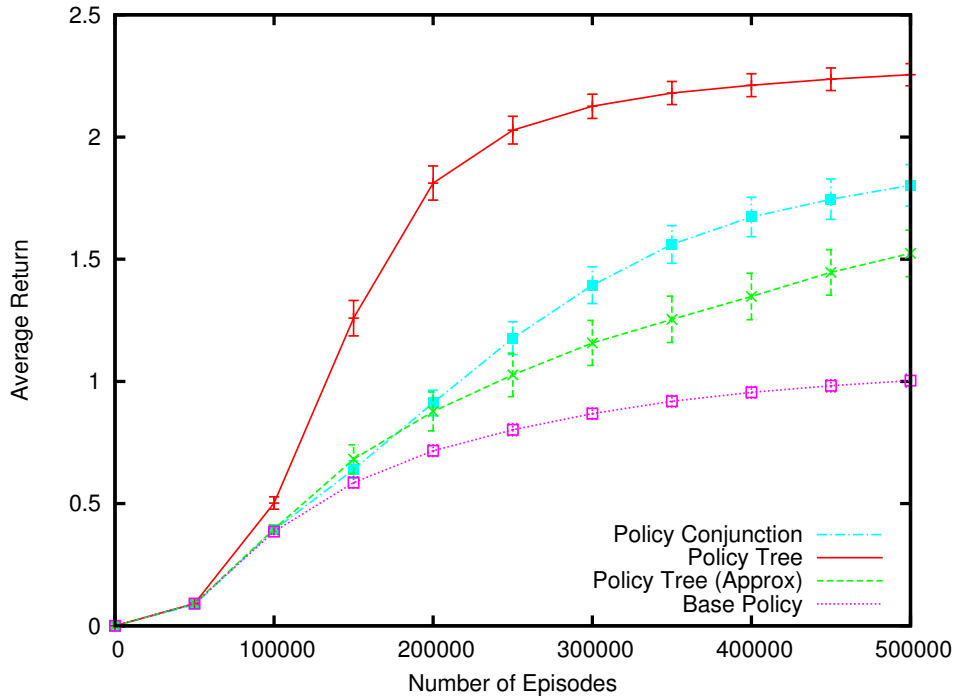

Figure 6.1: Results of the Policy Conjunction algorithm on Monsters

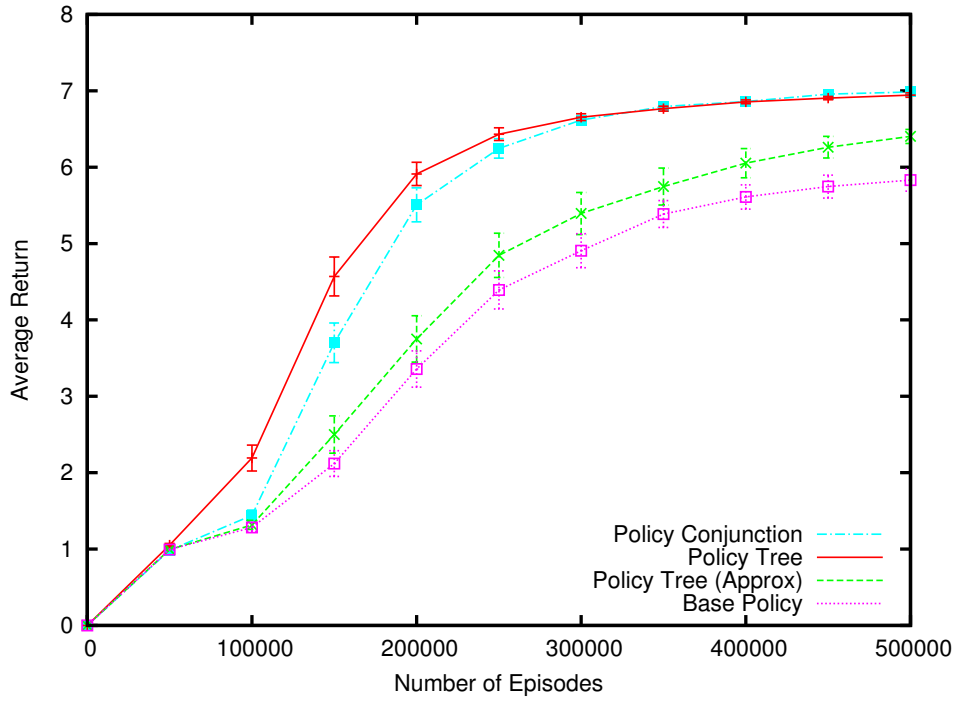

Figure 6.2: Results of the Policy Conjunction algorithm on Switcheroo

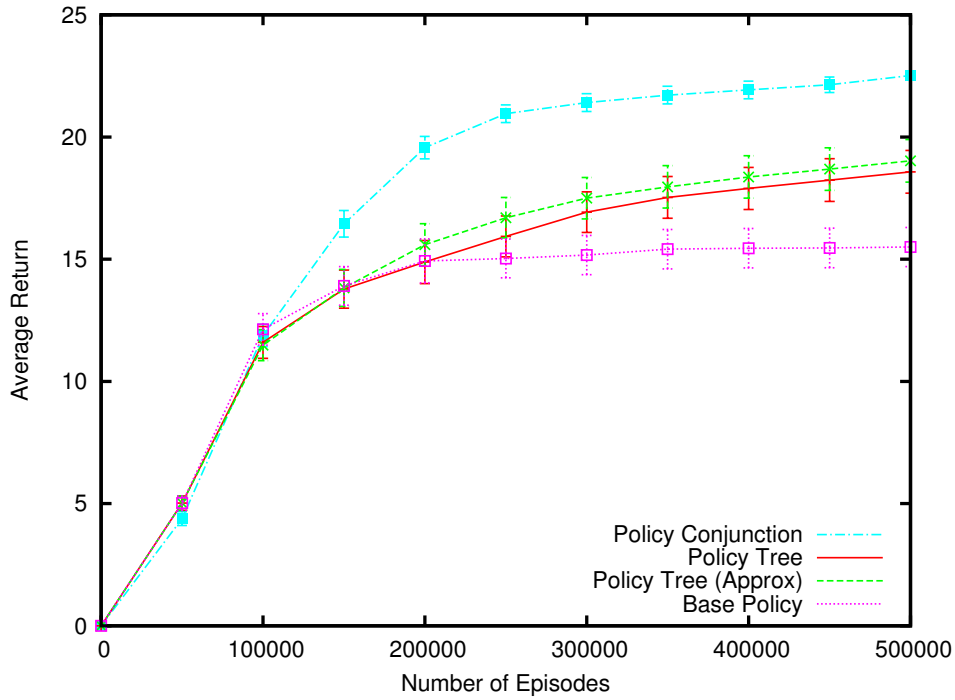

Figure 6.3: Results of the Policy Conjunction algorithm on Mothership

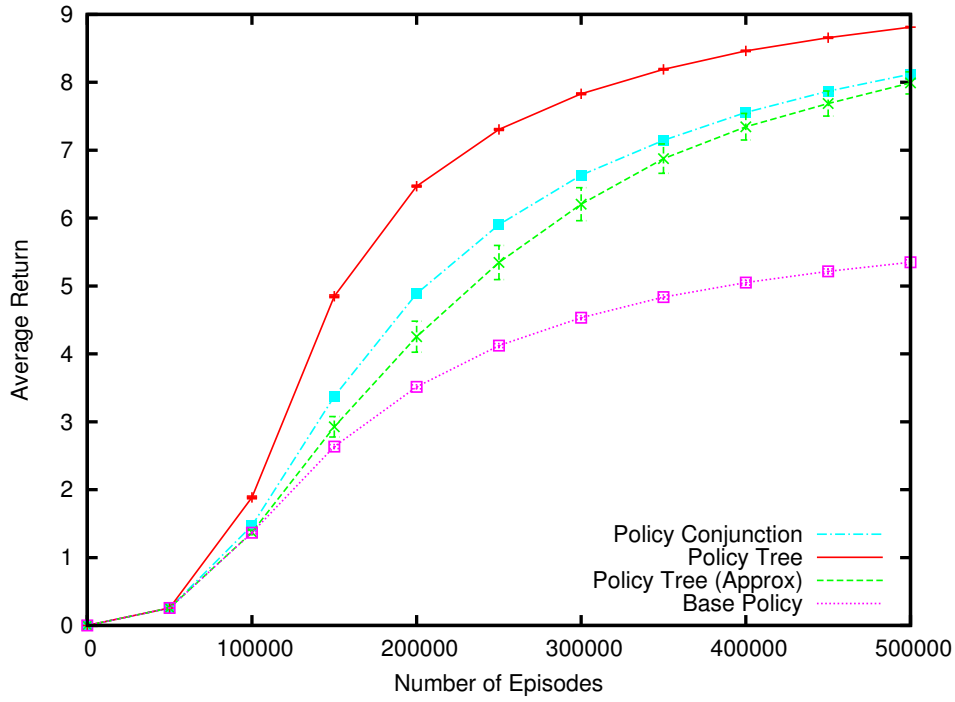

Figure 6.4: Results of the Policy Conjunction algorithm on Rescue

All of the parameters common to the two algorithms were kept identical. That is, a parameter optimization stage of 49000 episodes, a gradient averaging phase of 1000 episodes, a gradient norm of $q = 1$ during the growth phase, and a stopping condition parameter of $\lambda = 1$ were used. An initial step size equivalent to that used in the Policy Tree experiments was used for each domain, although it was decreased over time as described in Section 6.4. The number of features introduced in the growth phase was set to $C = D$, so that the computational complexity of the two algorithms was similar. And finally, as before, 30 different runs of learning over 500000 episodes were performed, and the average return was measured as the moving average of the total reward per episode with a window length of 50000.

The results show that the Policy Conjunction algorithm can significantly outperform the base policy. In general, Policy Conjunction tends to perform worse than Policy Tree during the early stage of learning, but can catch up with Policy Tree and even outperform it in some cases. The comparatively slower improvement over the base policy might be due to the decrease in step size after each growth phase in the former algorithm, which is necessary to ensure stability.

The domain where Policy Conjunction does comparatively worse is Monsters. As mentioned in Section 5.5, Policy Tree tends to split on the power feature (the presence of a power suit at the agent location, indicating whether the agent is powered) early in the learning process. This effectively creates a conjunction of this feature with every other feature, allowing the agent to learn completely different behaviours depending on the power feature (which is the desired policy). Policy Conjunction individually chooses a conjunction for each feature, and not all such decisions favour the power feature.

Policy Conjunction does well in Mothership and Rescue. In both of these domains, the algorithm outperforms Policy Tree by the end of the learning stage. Unlike in Monsters, Policy Tree does not always pick a good split in the beginning stages in these domains. Policy Conjunction, by retaining the existing features, most likely does not suffer too much from poor choices made in some of the runs.

Chapter 7

Future Work

There are a number of areas in which this work can be expanded. This chapter elaborates on a few of these.

7.1 Detecting Convergence of the Phases

Both the Policy Tree and Policy Conjunction algorithm use two phases for parameter optimization and representation growth, with fixed lengths. One could test for convergence of the objective function during optimization, to get the best possible performance before trying to expand the structure. However, highly optimized policies tend to be highly deterministic, making re-optimization of the parameters after a split trickier. The Gibbs policy, for instance, has a low gradient in regions of high determinism. The use of natural gradients (Kakade, 2001) could alleviate this problem, by measuring the change of the objective function with respect to the actual change in the policy distribution, rather than the change in parameter values.

7.2 Removing Redundant Features/Splits

Due to the presence of noise in the gradients, or due to the locality of our improvement measuring criterion, it is possible that some splits or conjunctive features do not enhance the policy significantly. This causes needless increase in the number of parameters, and slows down learning in both the Policy Tree and the Policy Conjunction algorithms. In the former, this would cause splitting the data available to each branch. A possible fix for this would be to prune the tree if optimization after a split does not significantly change the parameters. In the latter, additional

features decrease the step size, and could be removed by performing L1 regularization (Langford et al., 2009).

7.3 Generalizing the Decision Nodes in Policy Tree

The splits in the decision tree are currently based on the value of a single observation variable. In general, we could define a split over a conjunction or a disjunction of observation variables. This would increase the size of the fringe used during tree growth, but would allow the algorithm to find splits which may be significantly more meaningful. A different kind of split in the decision nodes would be on a linear combination of features. This can be viewed as a split on a non-axis aligned hyperplane in the observation space. As there are infinite such splits, it is not possible to measure them using our fringe structure. However, there may be ways to develop alternative criterion in order to grow a tree with such a representation. Prior research suggests that such an architecture is useful for classification but not for regression (Breiman et al., 1984). It is unclear how useful it would be in the search for optimal policy.

7.4 Using Off-Policy Experience

The two phase structure of our algorithm is slightly sample inefficient, as the experience during the gradient averaging phase is not used to optimize the parameters. In the experiments in this thesis, 2% of the episodes were unused for optimization. Due to the requirement to stay on-policy to estimate the gradient, this is difficult to avoid. One possible solution would be to use importance sample weighting and utilize off-policy trajectories to compute the gradient. This would in fact avoid the necessity of keeping the policy fixed to get a reliable gradient estimate. The use of importance sampling in policy gradient has been studied previously (Jie and Abbeel, 2010). However, the weights for the off-policy samples would reduce exponentially with the horizon of the problem, and it is uncertain whether it is possible to have a reliable off-policy estimate of the gradient in most domains.

Chapter 8

Conclusion

This thesis presented two algorithms for adaptive representation using policy gradient, and their utility was demonstrated on a variety of domains inspired by games. These algorithms have the same convergence guarantees as parametric policy gradient methods, but can adapt their representation whenever doing so improves the policy.

To the best of my knowledge, Policy Tree is the first algorithm which can learn a decision tree to represent the policy in a model-free setting. It is the most general of the algorithms presented, since it can work with any parametric representation of policy as its base.

In the case when the base policy depends only on linear functions of the features, two alternate approaches have been described. The first is an approximate version of Policy Tree, which has a linear computational complexity per time-step in the number of parameters during the entire learning phase, and can still improve over the base policy.

The second is the Policy Conjunction algorithm, which is an interesting variation of Policy Tree, demonstrating the relationship between decision tree methods and feature expansion. It is simpler to implement, and allows an arbitrary number of features to be introduced into the representation during the growth phase. It retains the original features of the policy, allowing it to correct poor choices of feature expansion made early in the algorithm. It improves upon Policy Tree in certain domains, although Policy Tree tends to perform better early during the learning process.

These algorithms demonstrate that the gradient of the expected return is a useful signal in the search for policy representation in reinforcement learning.

Bibliography

Au, M. and Maire, F. D. (2004). Automatic state construction using decision tree for reinforcement learning agents. International Conference on Computational Intelligence for Modelling Control and Automation.

Baxter, J. and Bartlett, P. L. (2000). Direct gradient-based reinforcement learning. In *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems*, volume 3, pages 271–274. IEEE.

Baxter, J. and Bartlett, P. L. (2001). Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15(1):319–350.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

Boyan, J. and Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. *Advances in Neural Information Processing Systems*, pages 369–376.

Breiman, L., Friedman, J., Stone, C. J., and Olshen, R. A. (1984). *Classification and regression trees*.

Da, Q., Yu, Y., and Zhou, Z.-H. (2014). Napping for functional representation of policy. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems*, pages 189–196.

Dabney, W. and McGovern, A. (2007). Utile distinctions for relational reinforcement learning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 7, pages 738–743.

Greensmith, E., Bartlett, P. L., and Baxter, J. (2004). Variance reduction techniques for gradient estimates in reinforcement learning. *The Journal of Machine Learning Research*, 5:1471–1530.

Jie, T. and Abbeel, P. (2010). On a connection between importance sampling and the likelihood ratio policy gradient. In *Advances in Neural Information Processing Systems*, pages 1000–1008.

Kakade, S. (2001). A natural policy gradient. *Advances in Neural Information Processing Systems*, 14:1531–1538.

Kersting, K. and Driessens, K. (2008). Non-parametric policy gradients: A unified treatment of propositional and relational domains. In *Proceedings of the 25th International Conference on Machine learning*, pages 456–463. ACM.

Kolmogorov, A. N. and Fomin, S. (1957). Elements of the theory of functions and functional analysis.

Langford, J., Li, L., and Zhang, T. (2009). Sparse online learning via truncated gradient. In *Advances in Neural Information Processing Systems*, pages 905–912.

McCallum, A. K. (1996). Learning to use selective attention and short-term memory in sequential tasks. In *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, volume 4, page 315. MIT Press.

Ng, A. Y., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., and Liang, E. (2006). Autonomous inverted helicopter flight via reinforcement learning. In *Experimental Robotics IX*, pages 363–372. Springer.

Peters, J. and Schaal, S. (2008a). Natural actor-critic. *Neurocomputing*, 71(7):1180–1190.

Peters, J. and Schaal, S. (2008b). Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697.

Puterman, M. L. (2009). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

Quinlan, J. R. (1993). *C4. 5: programs for machine learning*. Morgan kaufmann.

Singh, S. P., Jaakkola, T., and Jordan, M. I. (1994). Learning without state-estimation in partially observable markovian decision processes. In *Proceedings of the 11th International Conference on Machine Learning*, pages 284–292.

Sutton, R. S. and Barto, A. G. (1998). *Reinfocement Learning: An Introduction*. MIT Press.

Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12 (NIPS 1999)*, pages 1057–1063. MIT Press.

Uther, W. T. and Veloso, M. M. (1998). Tree based discretization for continuous state space reinforcement learning. In *Association for the Advancement of Artificial Intelligence*, pages 769–774.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.