University of the Witwatersrand
Computer Science Masters
Reinforcement Learning with Parameterized Actions

Warwick Masson
388188
Supervised by Pravesh Ranchod and George Konidaris

August 15, 2016

Declaration

I declare that this thesis is my own, unaided work. It is being submitted for the Degree of Master of Science at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University.

_____

(Signature of candidate)

_____ day of _____ 20 _____ in _____

Abstract

In order to complete real-world tasks, autonomous robots require a mix of fine-grained control and high-level skills. A robot requires a wide range of skills to handle a variety of different situations, but must also be able to adapt its skills to handle a specific situation. Reinforcement learning is a machine learning paradigm for learning to solve tasks by interacting with an environment. Current methods in reinforcement learning focus on agents with either a fixed number of discrete actions, or a continuous set of actions.

We consider the problem of reinforcement learning with parameterized actions—discrete actions with continuous parameters. At each step the agent must select both which action to use and which parameters to use with that action. By representing actions in this way, we have the high level skills given by discrete actions and adaptibility given by the parameters for each action.

We introduce the Q-PAMDP algorithm for model-free learning in parameterized action Markov decision processes. Q-PAMDP alternates learning which discrete actions to use in each state and then which parameters to use in those states. We show that under weak assumptions, Q-PAMDP converges to a local maximum. We compare Q-PAMDP with a direct policy search approach in the goal and Platform domains. Q-PAMDP out-performs direct policy search in both domains.

Acknowledgements

First, thanks to my parents for their support throughout this work. Although it was trying at times, they were always there for me when I needed them.

This thesis would not have been possible without all the work done by my supervisor George Konidaris. We worked closely together to turn what started as a broad idea into a focused piece of research. He encouraged me to submit this research to two conferences, which gave the work a vital stress-test.

Pravesh Ranchod, my other supervisor, deserves special thanks for helping me negotiate the processes towards finishing this Masters.

I am grateful to my fellow students for providing input and for being a sounding board for my ideas. It was extremely useful to be able to explain concepts to someone who was unfamiliar with my work. Thanks in particular to Dean Wookey for his help with editing, theory, and with the phrasing and formalization of the Q-PAMDP algorithm.

Contents

1 Introduction **6**

2 Background **8**
2.1 Reinforcement Learning 8
2.2 Value Functions 10
2.3 Policy Search 13
2.4 Summary 15

3 Reinforcement Learning with Parameterized Actions **17**
3.1 Naive Approaches 19
3.2 Related Work 20
3.3 Summary 22

4 The Q-PAMDP Algorithm **23**
4.1 Convergence Properties of Q-PAMDP(1) 24
4.2 Gradient of $H_M(\theta)$ 26
4.3 Convergence properties of Q-PAMDP(∞) 27
4.4 Summary 27

5 The Robot Soccer Goal Domain **28**
5.1 Domain 29
5.2 Implementation Details 31
5.3 Results 32

6 The Platform Domain **34**
6.1 Specification 34
6.2 Approach 35
6.3 Results 36
6.4 Summary 37

7 Conclusions and Future Work **38**
7.1 Future Work 38
7.2 Conclusion 38
References 41

Appendices **42**

A Theoretical Definitions **43**

B Disconnected Action Spaces **44**

C Discontinuous Policies **45**

D RSS Simulator **46**

List of Figures

1.1 Parameterized actions for a soccer player robot. 7

2.1 Interaction between an agent and the environment. 8
2.2 A simple maze problem. 9
2.3 Discrete and continuous state representations 10
2.4 Policies for discrete and continuous state spaces. 11
2.5 Discrete and continuous action spaces. 12
2.6 Examples of 2-dimensional Fourier basis functions. 12
2.7 Gradient ascent on a surface. 14

3.1 A quad-rotor delivery system. There are two actions: move-to(x, y, z), which moves the quadrotor to position (x, y, z), and drop-payload(x, y), which drops the payload at position (x, y). 17
3.2 Discrete, continuous, and parameterized action spaces. 18
3.3 Action selection with a two-tier policy. 19
3.4 The WAM robot arm. 19

5.1 The robot soccer 2D simulator. 28
5.2 The robot soccer goal domain. 29
5.3 Average return in the goal domain. 32
5.4 Average goal scoring probability. 33
5.5 A converged robot soccer goal episode. 33

6.1 A screenshot from the Platform domain. 34
6.2 The components of the Platform domain. 35
6.3 The Platform domain's actions: *run*, *hop*, and *leap*. 35
6.4 Average percentage distance covered in the Platform domain. 36
6.5 A successful episode of the Platform domain. 37

B.1 The ball sorting problem: balls must be dropped into one of the boxes, but not in between them. 44

C.1 A discontinuous policy domain. 45
C.2 Optimal policy $\pi^*(x)$ for the discontinuous policy domain. 45

Chapter 1

Introduction

As research in robotics and autonomous control advances, efforts are being made to tackle complex, real-world problems. These problems are too difficult to solve with a fixed, manually specified algorithm, and require agents that learn from experience. Reinforcement learning is a field that is interested in learning solutions for control problems.

In reinforcement learning, we typically consider either a discrete or a continuous action space [Sutton and Barto 1998]. With a discrete action space, we make decisions about which distinct action we wish to perform from a finite action set. With a continuous action space, we express the selected action as a real-valued vector. If we use a continuous action space, we lose the ability to consider differences in kind: all actions must be expressible as a single vector. However, if we only use discrete actions, we either lose the ability to apply the same action with different parameters, or suffer a blow-up in the number of actions as we must represent variations of the same action as distinct.

A parameterized action is a discrete action parameterized by a real-valued vector. Modeling actions this way introduces structure into the action space by treating different kinds of continuous actions as distinct. At each step an agent must choose both which action to execute based on the state and what parameters to execute it with.

For example, consider a soccer playing robot that can kick, pass, or run. We can associate a continuous parameter vector to each of these actions: we can kick the ball to a target with a given force, pass to another player at a specific position, or dribble to another position. This is illustrated in figure 1.1. Each of these actions has its own distinct parameter vector.

This thesis introduces parameterized action Markov decision processes (PAMDPs), which model reinforcement learning problems with either distinct actions which require parameters to adjust the action to different situations, or where there are multiple mutually incompatible continuous actions. Each of these actions is parameterized in its own way, so expressing an action as a single vector containing all the parameters for each of these movements would be redundant, as only a small subset of the parameters are used for any action.

Although prior research has included parameterized and hybrid discrete-continuous actions, it has only done so from a planning perspective. Additionally, such work has largely assumed the parameter set is the same for all actions [Hoey *et al.* 2013; Rachelson 2009; Guestrin *et al.* 2004]. We consider the problem from a reinforcement learning perspective where we have no information about the model or reward function, and allow for each action to have its own set of parameters.

We focus on how to learn an action-selection policy given pre-defined parameterized actions. We in-

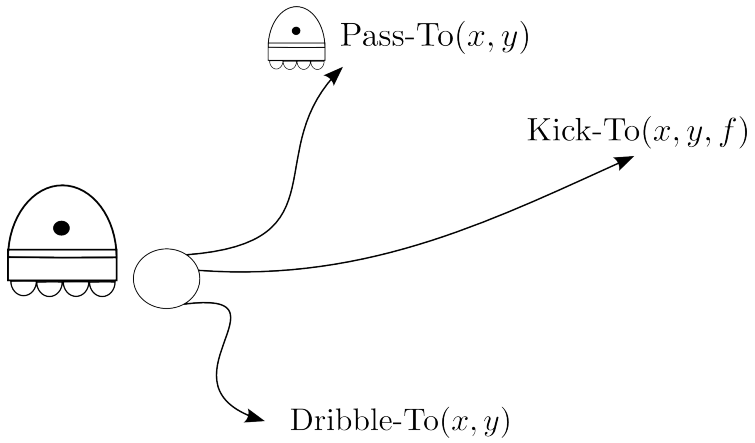


Figure 1.1: Parameterized actions for a soccer playing robot: dribble, kick, and pass.

troduce the Q-PAMDP algorithm, which alternates learning action-selection and parameter-selection policies and compare it to a direct policy search method. We show that with appropriate update rules Q-PAMDP converges to a local maximum. These methods are compared empirically in the goal and Platform domains, where Q-PAMDP out-performed direct policy search and fixed parameter SARSA.

This thesis is organized as follows. In chapter 2 we explain Markov decision processes, reinforcement learning, and policy search. In chapter 3 we introduce the idea of parameterized actions and define the parameterized action MDP. Chapter 4 details the Q-PAMDP algorithm and provides theoretical results pertaining to it. Chapter 5 and 6 give the experimental results for the goal domain and Platform domain, respectively. Chapter 7 concludes the thesis and covers related research, and discusses potential future work on this topic.

Chapter 2

Background

In this chapter we survey reinforcement learning and policy search. The first section considers reinforcement learning, the agent and environment model, and Markov decision processes. Next, we discuss the value and action-value functions. We then consider policy search, looking at the episodic natural actor-critic (eNAC) algorithm in particular.

2.1 Reinforcement Learning

In reinforcement learning, we are concerned with solving control problems with a reward signal. The control problem involves an agent taking actions based on its state, after which it receives a new state and a reward. We want to learn which actions to take to maximize not just the next reward, but the future reward. This process is illustrated in figure 2.1.

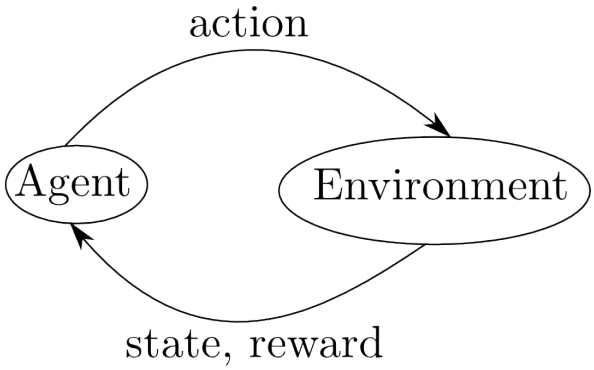


Figure 2.1: A single interaction between an agent and the environment. At each step, the agent selects an action and executes it, receiving a new state and a reward.

As an example of a reinforcement learning problem, we might have a robot (the agent) with a location (the state), that is navigating a maze (the environment). This is depicted in figure 2.2. The robot can move forward, left, right, or back (the actions) and receives a number upon taking a transition indicating the cost of taking a single step, or a large reward for reaching the goal. Our task is to design an algorithm that will learn a policy that maps from states to actions, essentially determining how the agent acts in each state. To return to the maze example, a successful policy would tell the robot which direction to move in each state so as to finish the maze.

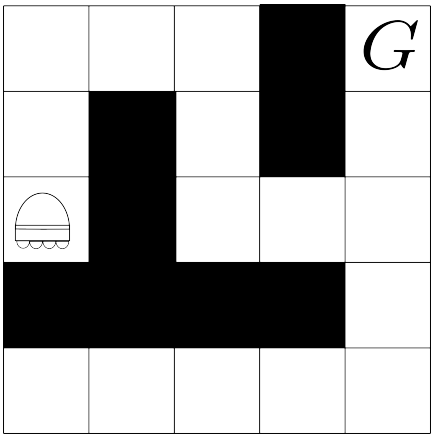


Figure 2.2: A simple maze problem. The robot (middle left) must navigate the maze to find the goal (marked G).

We can formalize this problem as a Markov decision process. A Markov decision process (MDP) is a tuple $\langle S, A, P, R, \gamma \rangle$, where S is a set of states, A is a set of actions, $P(s, a, s')$ is the probability of transitioning to state s' from state s after taking action a, $R(s, a, r)$ is the probability of receiving reward r for taking action a in state s, and γ is a discount factor [Sutton and Barto 1998]. An agent interacts with an MDP by repeatedly taking an action and receiving a state and a reward.

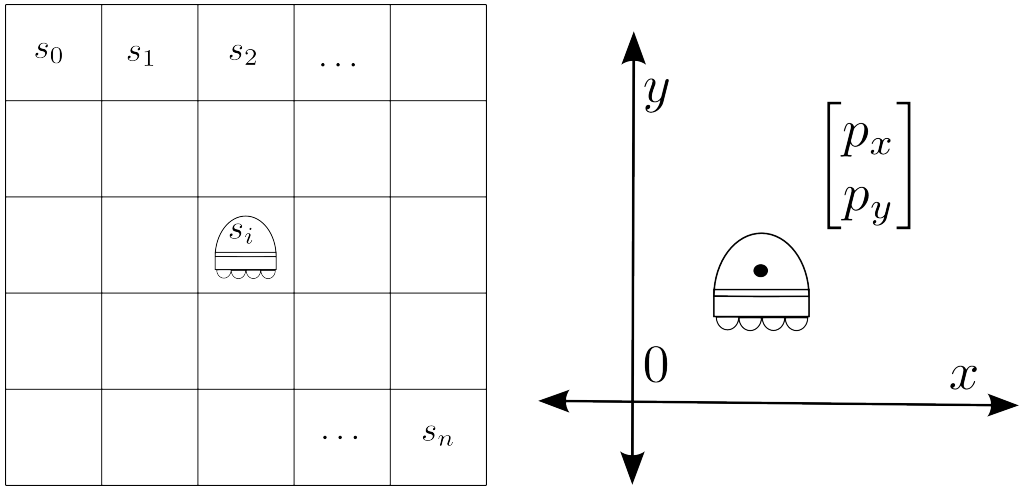
The solution to an MDP is a policy π. A policy can be either stochastic or deterministic. A deterministic policy π is a mapping $\pi : S \to A$ which selects an action for each state. A stochastic policy $\pi(a|s)$ is a distribution which the determines the probability of selecting action a in state s. We wish to find a policy which maximizes the expected sum of discounted rewards (the return). We are interested in stochastic policies in this dissertation as they are more flexible and easier to optimize.

A state space can be either discrete or continuous. A discrete state space has a finite set of states $S = \{s_1, \ldots, s_n\}$. A continuous state space has an infinite set of states, $S \subseteq \mathbb{R}^n$. Consider a two-dimensional robotics problem, where the robot has an x and a y position. With a discrete representation there are a finite number of completely distinct possible positions, as depicted in figure 2.3a. In a continuous representation the state is given by a 2-dimensional value, as depicted in figure 2.3b.

How we represent a policy depends on the state space. A discrete state space requires a different choice for each state (figure 2.4a). A continuous state space policy is based on a continuous function over the state space (figure 2.4b).

Similarly, the action space can be either discrete or continuous. A discrete action space is a finite set of actions $A = \{a_1, \ldots, a_n\}$. To continue with our robotics example the robot may the move in different cardinal directions, as depicted in figure 2.5a. With a continuous action space the robot selects a real-value vector from an action space $A \subseteq \mathbb{R}^m$. The robot can move with force (f_x, f_y), as depicted in figure 2.5b. Note that a continuous state space can have a discrete or a continuous action space.

A model of an MDP contains information about its transition and reward functions. In a planning problem we have an accurate model of the MDP, and we must compute the policy using this knowledge. In reinforcement learning we do not have access to the MDP but we can interact with it. Our approach can either be model-based or model-free. With model-based reinforcement learning, the agent constructs a model of the MDP using its experience of the MDP. A model represents the transition and reward functions of the MDP. We are concerned with model-free learning, where we learn without constructing a model.



(a) Discrete state space.

(b) Continuous state space.

Figure 2.3: Discrete and continuous state representations for the position of a robot. The discrete state space is a finite set of distinct positions. The continuous state space is a subset of all real-valued positions.

2.2 Value Functions

The value function $V^\pi(s)$ is defined as the expected discounted return achieved by policy π starting at state s:

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \right],$$

where $\mathbb{E}_x [y]$ denotes the expected value of y given x. In other words, this is the expected return over trajectories starting at state s and following policy π. Similarly, we define the action-value function

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[r_0 + \gamma V^\pi(s) \right],$$

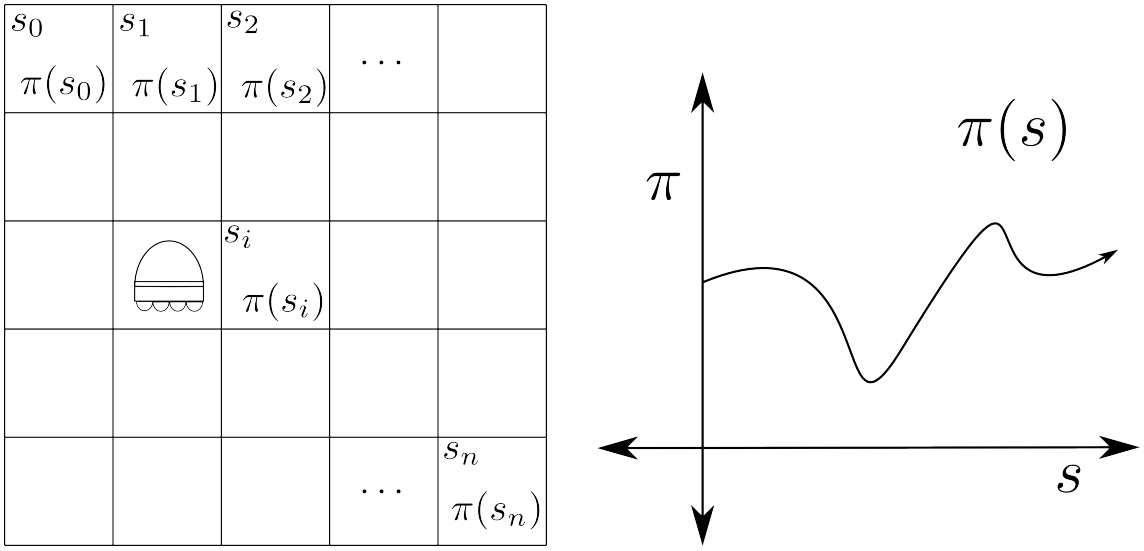
as the expected return r_0 obtained by taking action a in state s, and then following policy π thereafter. If we want to use a value function for control, we require a model. This is because to determine the best action for a state s, the best action is the one which maximizes $V(s')$. To know the value of action a, we need to know what s' will be (or likely be) after taking action a.

To avoid this requirement, we learn the values of each action independently. Knowing the action-value function allows us to select the action which maximizes $Q^\pi(s, a)$. We can learn Q for an optimal policy using a method such as SARSA [Sutton and Barto 1998].

With a discrete state space, we can represent Q with a table that contains the value for each action-value pair. SARSA (algorithm 1) can learn the values for such a tabular representation. SARSA is an on-policy algorithm, which means that it bases its policy on the current action-value function while learning it. The algorithm uses a temporal difference update rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right].$$

which is analogous to the temporal difference update rule for $V(s)$. This rule updates the action-value $Q(s_t, a_t)$ with the reward r_t and the value of our next action $Q(s_{t+1}, a_{t+1})$ for the next step. This accounts for the current reward and the expected future reward.



(a) Discrete state space policy.

(b) Continuous state space policy.

Figure 2.4: Deterministic policies for discrete and continuous state spaces.

Algorithm 1 SARSA [Sutton and Barto 1998]

Algorithm:
$s \leftarrow$ initial state
$a \sim \pi(a|s)$
repeat
  Take action a, receive reward r, state s'
  $a' \leftarrow \pi(a'|s')$
  $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$
  $s \leftarrow s'$
  $a \leftarrow a'$
until Q converges

In domains with a continuous state space, we can represent $Q(s,a)$ using parametric function approximation with a set of parameters ω. The simplest form of parametric function approximation is linear function approximation:

$$Q_\omega(s,a) = \omega^T \phi_a(s),$$

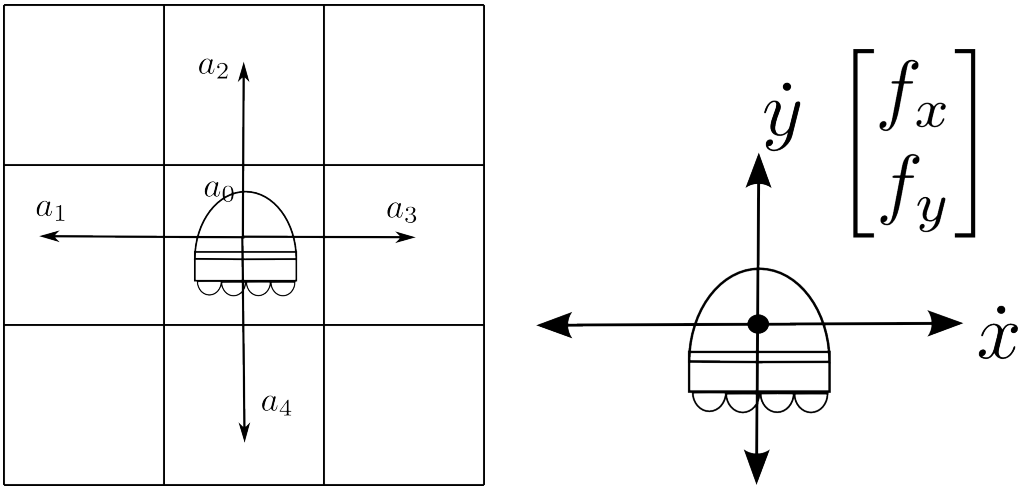
where ω is a parameter vector and ϕ_a is a feature vector for action a.

One basis scheme (which we adopt in this thesis) is the Fourier basis function, which uses features:

$$\phi_j(s) = \cos(\pi c_j \cdot s),$$

where c_j is a frequency vector with each element an integer between 0 and d, and d is the degree of the Fourier basis. Figure 2.6 depicts several example Fourier basis functions. The basis is obtained by enumerating all such frequency vectors. This results in $(d+1)^n$ basis functions for an order n Fourier basis. We can reduce this number by placing restrictions on the coupling of coefficients used in the vectors c_j. The idea behind this basis is Fourier series approximation: for a function f with period T, it can be approximated with the series

$$\bar{f}(x) = \frac{a_0}{2} + \sum_{k=1}^{n} \left[a_k \cos\left(k\frac{2\pi}{T}x\right) + b_k \sin\left(k\frac{2\pi}{T}x\right)\right].$$



(a) Discrete action space. (b) Continuous action space.

Figure 2.5: Discrete and continuous action spaces. The discrete action space is a finite set of distinct directions, while the continuous action space is a real-valued powers.

Where a_k, b_k are coefficients. As the function we are approximating is unknown, we must determine the coefficients a_k. If we scale the coefficients between 0 and 1, we can assume that the function is periodic over $[-1, 1]$ (period 2), and is an even function i.e. $b_k = 0$. This leaves us with an approximation

$$\bar{f}(x) = \frac{a_0}{2} + \sum_{k=1}^{n} a_k \cos(2\pi k x),$$

which gives us the basis functions used above [Konidaris *et al.* 2011].

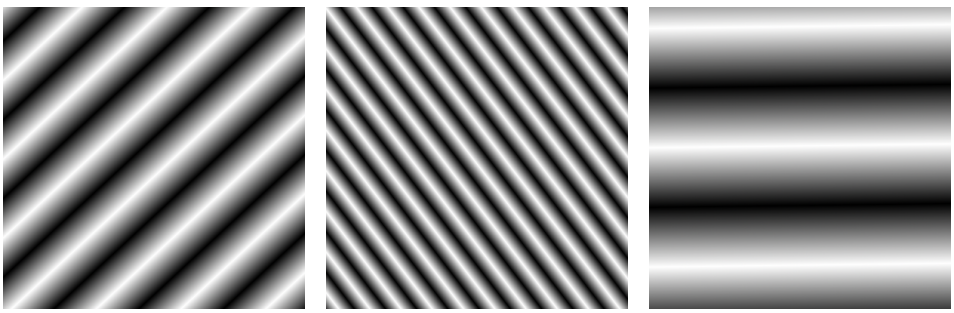


Figure 2.6: Examples of 2-dimensional Fourier basis functions.

Algorithms such as gradient descent SARSA [Sutton and Barto 1998] learn this representation of Q. The idea is that we can perform a gradient update, which takes the form

$$\omega_{t+1} = \omega_t + \alpha_t \left[r_{t+1} + \gamma Q_{\omega_t}(s_{t+1}, a_{t+1}) - Q_{\omega_t}(s_t, a_t) \right] \nabla_{\omega_t} Q_{\omega_t}(s_t, a_t).$$

This is analogous to discrete SARSA but we update Q with the gradient instead of updating a single entry in a table. If we use linear function approximation, the gradient is simply

$$\nabla_{\omega_t} Q_{\omega_t}(s_t, a_t) = \phi(s_t, a_t),$$

where $\nabla_x f(x)_i = \partial f(x)/\partial x_i$.

Algorithm 2 Gradient Descent SARSA(λ) [Sutton and Barto 1998]

Input:
Action-Value representation $Q_\omega(s, a)$
Policy π dependent on Q_ω
Algorithm:
Initialize ω randomly
$s \leftarrow$ initial state
$a \sim \pi(a|s)$
repeat
  Take action a, receive reward r, state s'
  if s' is terminal **then**
    $\delta \leftarrow r - Q_\omega(s, a)$
  else
    $a' \sim \pi(a'|s')$
    $\delta \leftarrow r + \gamma Q_\omega(s', a') - Q_\omega(s, a)$
  end if
  $e \leftarrow \gamma\lambda e + \nabla_\omega Q_\omega(s, a)$
  $\omega \leftarrow \omega + \alpha\delta e$
  $s \leftarrow s'$
  $a \leftarrow a'$
until θ converges

To speed up the learning process, we can uses traces. The idea behind traces is that the reward can be attributed not just to the current state and action but also previous ones. By doing so, we can propagate rewards back to an action that led to a useful state. SARSA(λ) uses traces with a trace factor λ. The trace factor determines to what degree rewards can be attributed to previous states. Algorithm 2 gives the full description of gradient descent SARSA(λ).

2.3 Policy Search

For problems with a continuous action space ($A \subseteq \mathbb{R}^m$) we encounter the action selection problem: selecting the optimal action with respect to $Q(s, a)$ is non-trivial, as it requires finding a global maximum for a function in a continuous space. We can avoid this problem by using a policy search algorithm, given a class of policies parameterized by a set of parameters θ. A parameterized policy π_θ can directly return a continuous action and so avoids searching the entire action space at each step. This transforms the problem into one of direct optimization over θ. Several approaches to policy search exist, including policy gradient methods, entropy-based approaches, path integral approaches, and sample-based approaches [Deisenroth *et al.* 2013]. We focus on gradient methods due to their relative simplicity and useful convergence properties.

Policy gradient methods use gradient ascent to optimize the parameters. Gradient ascent works by updating the parameters of a function in the direction of fastest increase of that function. This direction is given by the gradient of the function with respect to its parameters. Figure 2.7 depicts this method. The idea is that if $J(\theta)$ represents the expected discounted return for policy π_θ, and J is differentiable with respect to θ, then we can update θ with

$$\theta_{t+1} = \theta_t + \alpha_t \nabla_\theta J(\theta), \tag{2.1}$$

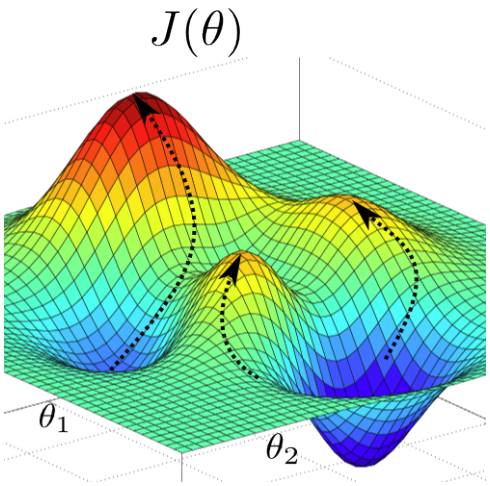


Figure 2.7: Gradient ascent on surface $J(\theta)$ with respect to $\theta = (\theta_1, \theta_2)$. As gradient ascent is a local optimization method, the maximum it converges to depends on the initial parameters.

where α_t is a decreasing update rate. This works because $\nabla_\theta J(\theta)$ is the direction of the fastest increase in $J(\theta)$ with respect to θ. As long as α_t is sufficiently small, $J(\theta_{t+1}) \geq J(\theta_t)$ (or this holds in the expectation). This is a local optimization method, so it will convergence to some local optima θ^* with respect to $J(\theta)$.

The difficulty lies in estimating $\nabla_\theta J(\theta)$ [Deisenroth *et al.* 2013]. The policy gradient is given by

$$\nabla_\theta J(\theta) = \int_\tau \nabla_\theta p_\theta(\tau) R(\tau) d\tau, \tag{2.2}$$

where τ is a trajectory, $R(\tau)$ is the return for τ, and $p_\theta(\tau)$ is the probability of encountering τ under policy π_θ [Deisenroth *et al.* 2013]. Using the identity

$$p_\theta(\tau) = p(s_0) \prod_{t=0}^{STEPS-1} p(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t) \tag{2.3}$$

$$\tag{2.4}$$

which follows as $p_\theta(\tau)$ is the combined probability of the entire sequence and as

$$\log p_\theta(\tau) = \log p(s_0) + \sum_{t=0}^{STEPS-1} \log p(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t) \tag{2.5}$$

$$\nabla_\theta \log p_\theta(\tau) = \sum_{t=0}^{STEPS-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{2.6}$$

we can obtain the identity

$$\nabla_\theta p_\theta(\tau) = p_\theta(\tau) \nabla_\theta \log p_\theta(\tau). \tag{2.7}$$

With these identities we can write the gradient as the expectation

$$\nabla_\theta J(\theta) = \mathbb{E}_{p_\theta(\tau)} \left[\nabla_\theta \log p_\theta(\tau) R(\tau)\right] \tag{2.8}$$

$$= \mathbb{E}_{p_\theta(\tau)} \left[\sum_{t=0}^{STEPS-1} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right], \tag{2.9}$$

as taking the log allows us to turn the product into a sum, and remove the dependence on the transition probabilities $p(s_{t+1}|s_t, a_t)$ [Deisenroth *et al.* 2013].

With a standard gradient, we assume that all parameters have a similar effect on the objective function. If we do not make this assumption, then differences between $p_\theta(\tau)$ and $p_{\theta+\Delta\theta}(\tau)$ can be large [Deisenroth *et al.* 2013]. The change caused by each parameter is measured by the Kullback-Leibler divergence. The Fisher information matrix given by

$$F_\theta = \mathbb{E}_{p(\tau)}\left[\nabla_\theta \log p_\theta(\tau) \nabla_\theta \log p_\theta(\tau)^T\right].$$

can be used to approximate the Kullback-Leibler divergence. By inverting this matrix, we can obtain the natural gradient

$$\nabla_\theta^{NG} J(\theta) = F_\theta^{-1} \nabla_\theta J(\theta),$$

where each parameter has a similar effect on J [Deisenroth *et al.* 2013].

The episodic natural actor critic (eNAC) algorithm, given in algorithm 3, computes a natural gradient using n sample paths τ [Peters and Schaal 2008]. The eNAC algorithm is intended for a finite horizon domain and as such we optimize the expected sum of rewards without discounting. The policy gradient is given as

$$\nabla_\theta^{\text{eNAC}} J(\theta) = F_\theta^{-1} \nabla_\theta J(\theta) \tag{2.10}$$

$$= \mathbb{E}_{p_\theta(\tau)}\left[\psi\psi^T\right] \mathbb{E}_{p_\theta(\tau)}\left[\psi R(\tau)\right]. \tag{2.11}$$

where

$$\psi = \sum_{t=0}^{STEPS-1} \nabla_\theta \log \pi_\theta(a_t|s_t).$$

We can compute these expectations by taking the average over runs. This kind of evaluation tends to be very noisy, which results in our estimate of the gradient having a large variance. By introducing the baseline b for reward:

$$\nabla_\theta J_\theta = \mathbb{E}_{p_\theta(\tau)}\left[\nabla_\theta \log p_\theta(\tau)(R(\tau) - b)\right],$$

we can reduce the variance by selecting the appropriate baseline [Deisenroth *et al.* 2013]. The baseline does not bias the gradient estimate, as

$$\mathbb{E}_{p_\theta(\tau)}\left[\nabla_\theta \log p_\theta(\tau) b\right] = b \int_\tau \nabla_\theta p_\theta(\tau) d\tau \tag{2.12}$$

$$= b \nabla_\theta \int_\tau p_\theta(\tau) d\tau \tag{2.13}$$

$$= b \nabla_\theta 1 \tag{2.14}$$

$$= 0. \tag{2.15}$$

We choose b so as to minimize the variance of the gradient estimate. The optimal b depends on the particular method of estimating the gradient. The eNAC algorithm uses a vector $\phi(s_0)$ to compute the baseline b [Deisenroth *et al.* 2013].

2.4 Summary

We have covered reinforcement learning, value functions, Q-learning and function approximation, and policy search. All these methods assume the agent is equipped with discrete or continuous actions. In the next chapter we introduce a formalization of the reinforcement learning problem that includes both.

Algorithm 3 Episodic Natural Actor Critic (eNAC)

Input:
Parameterized policy π_θ
Initial parameters θ
Initial-feature function ϕ
Algorithm:
for $i = 1$ **to** N **do**
Collect samples: $s^{[i]}_{1:STEPS}$, $a^{[i]}_{1:STEPS-1}$, $r^{[i]}_{1:STEPS}$

$$R^{[i]} = \sum_{t=1}^{STEPS} r_t^{[i]}$$

$$\psi^{[i]} = \begin{bmatrix} \sum_{t=1}^{STEPS-1} \nabla_\theta \log \pi_\theta \left(a_t^{[i]} | s_t^{[i]} \right) \\ \phi(s_0^{[i]}) \end{bmatrix}$$

end for

$$R = \left[R^{[1]}, \ldots, R^{[N]} \right]^T$$
$$\Psi = \left[\psi^{[1]}, \ldots, \psi^{[N]} \right]$$
$$w = ((\Psi^T \Psi)^{-1} \Psi^T R)[1:N]$$

return $\nabla_\theta^{\text{eNAC}} J(\theta) = w$

Chapter 3

Reinforcement Learning with Parameterized Actions

A parameterized action is a discrete action that takes a continuous set of parameters. Such an action consists of two parts: the discrete action and the continuous parameters. Intuitively, the discrete part determines the kind of action (the "what"), while the parameters determine its application (the "how").

For example, a quad-rotor delivery system might have a move action, move-to(x, y, z), and an action drop-payload(x, y). Figure 3.1 depicts this domain. Each of these actions is completely distinct from the other, and we would not want to combine these actions into a single continuous action lest we experiment with dropping payloads while moving.

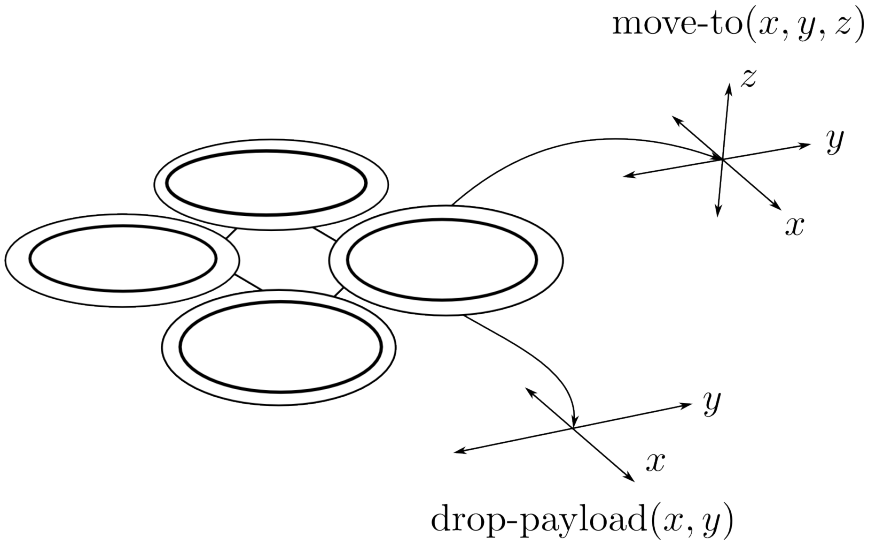


Figure 3.1: A quad-rotor delivery system. There are two actions: move-to(x, y, z), which moves the quadrotor to position (x, y, z), and drop-payload(x, y), which drops the payload at position (x, y).

We consider MDPs where the state space is continuous ($S \subseteq \mathbb{R}^n$) and the actions are parameterized. By this, we mean there is a finite set of actions $A_d = \{a_1, a_2, \ldots, a_k\}$, and for each action $a \in A_d$, there is a set of continuous parameters $X_a \subseteq \mathbb{R}^{m_a}$. The action space is then given by

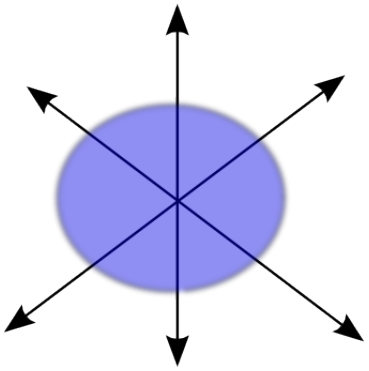
$$A_{d,x} = \bigcup_{a \in A_d} \{(a, x) \mid x \in X_a\}. \tag{3.1}$$

We refer to such MDPs as parameterized action MDPs (PAMDPs). Figure 3.2 depicts the different action spaces: discrete, continuous, and parameterized.

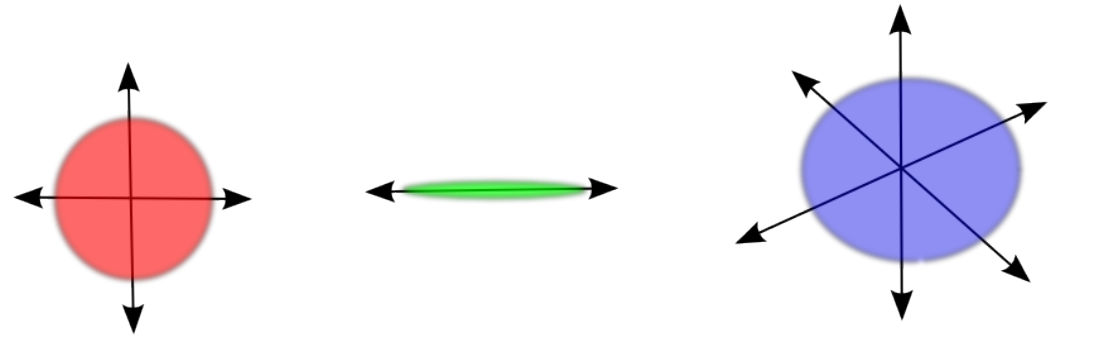
$$\{a_1, a_2, \ldots, a_m\}$$

(a) The discrete action space consists of a finite set of distinct actions.

$$X \subseteq \mathbb{R}^m$$



(b) The continuous action space is a single continuous real-valued space.



(c) The parameterized action space has discrete actions, each of which has a continuous action space.

Figure 3.2: The three types of action spaces: discrete, continuous, and parameterized.

We apply a two-tiered approach for domains with parameterized actions: selecting the parameterized action, and selecting the parameters for that action. We denote the action policy by $\pi^d(a|s)$. To select the parameters for an action a, we denote the action-parameter policy as $\pi^a(x|s)$. The policy is then given by $\pi(a, x|s) = \pi^d(a|s)\pi^a(x|s)$. In other words, to select a complete action (a, x), we first sample a discrete action a from $\pi^d(a|s)$ and then sample a parameter x from $\pi^a(x|s)$. Figure 3.3 depicts this selection process.

Consider a robotics domain where we can apply a torque at each joint in a seven jointed robot arm, as depicted in figure 3.4. An action can be specified as a torque to each joint so that

$$a = [\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7]^T .$$

This gives us an action space $A \subseteq \mathbb{R}^7$. Working with such a low-level high-dimensional action space is difficult, so learning high-level low-dimensional skills is preferable. A skill is a multi-step or complex

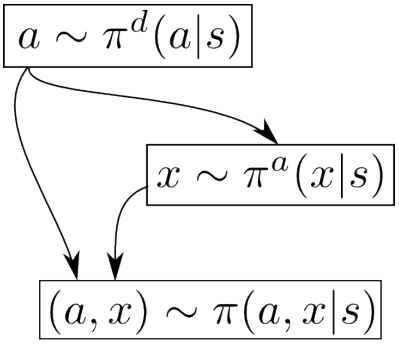


Figure 3.3: Action selection with a two-tier policy. At each step the agent first selects action a using the discrete policy π^d, then selects parameters x using parameter-policy π^a. The complete action is then (a, x).

action that abstracts the low-level motor functions. For example, we may learn a skill for positioning the end-effector at a position (x, y, z), or a skill for moving an object to a position (x, y). A low level skill is one with a low number of control parameters (in this case, the torques). Each skill would translate the control into the appropriate low-level torques. As such, we could not combine skills as they would each indicate different torques. Therefore we could treat each skill as a parameterized action.

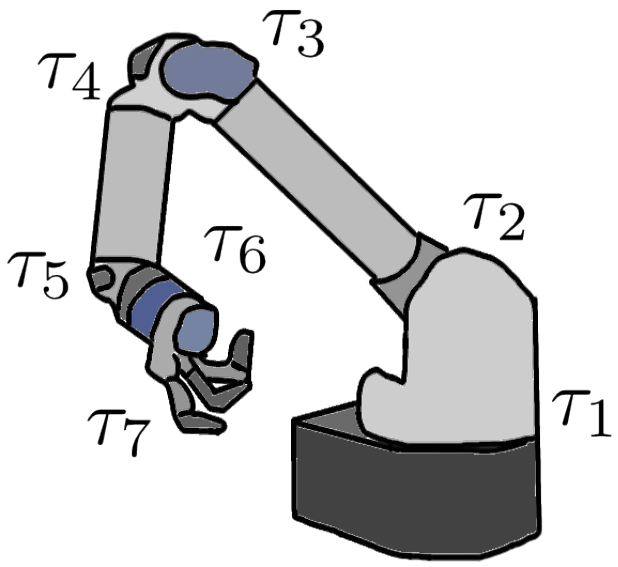


Figure 3.4: The WAM robot arm, with 7 torques.

PAMDPs can also be used to represent disconnected action spaces (appendix B) or discontinuous action spaces (appendix C).

3.1 Naive Approaches

One approach for handling a continuous action space is to select a finite number of vectors from that space and use the set of these vectors as a discrete action space. We refer to this approach as discretization. For a continuous action space $A \subseteq \mathbb{R}^m$, we select actions $a_1, a_2, \ldots, a_k \in A$. Our discretized

action space becomes $A_d = \{a_1, a_2, \ldots, a_k\}$. There are different schemes for selecting these actions.

The simplest is to evenly space actions over the action space. As an example, suppose we had a continuous action space $A = [-1, 1]$. One discretization would be $A_d = \{-1, -0.5, 0, 0.5, 1\}$. We can increase the granularity by using more actions, so that

$$A_d = \{-1, -0.75, -0.5, -0.25, 0, 0.25, 0.5, 0.75, 1\},$$

and so on. If we have a multi-dimensional action space, the number of actions needed to cover it will increase exponentially with the number of dimensions.

We can apply this approach by discretizing each parameter space individually. If we have discrete actions $a_1, \ldots, a_k$ with parameter spaces $X_1, \ldots, X_k$, then for each X_i select a set of parameters

$$X_i^d = \{x_{i1}, x_{i2}, \ldots, x_{il_i}\} \subseteq X_i.$$

Then the complete discretized action space is given by

$$A = \{(a_1, x_{11}), \ldots, (a_1, x_{1l_1}), (a_2, x_{21}), \ldots, (a_2, x_{2l_2}), \ldots, (a_k, x_{k1}), \ldots, (a_k, x_{kl_k})\}.$$

The drawback of using discretization is adequately representing the action space may require many discrete actions. This can make learning prohibitively expensive. Discretization is an inappropriate approach for such action spaces.

The next approach we consider is one of direct policy search. The action policy is defined by the parameters ω and is denoted by $\pi_\omega^d(a|s)$. The action-parameter policy for action a is determined by a set of parameters θ_a, and is denoted $\pi_\theta^a(x|s)$. Let $\Theta = (\theta, \omega)$. We use a direct policy search method to optimize

$$J(\Theta) = \mathbb{E}_{s_0 \sim D}\left[V^{\pi_\Theta}(s_0)\right], \tag{3.2}$$

with respect to Θ, where s_0 is an initial state sampled according to the initial state distribution D. J is the expected return for a given policy starting at an initial state. Note that were previously J was a function of θ when discussing policy optimization, now J is a function of all parameters Θ. The disadvantage with this approach is that direct optimization works poorly on policies with large numbers of parameters, and we require a large number of variables to represent the action policy.

Policy gradient algorithms (eNAC for example) may require computation of the log policy gradient features, given by $\nabla_\Theta \log \pi_\Theta(a, x|s)$. To compute these features for a two-tier policy, note that

$$\begin{aligned}
\nabla_\theta \log \pi_\Theta(a, x|s) &= \nabla_\theta \log \pi_\omega^d(a|s)\pi_\theta^a(x|s) && (3.3)\\
&= \nabla_\theta \log \pi_\theta^a(x|s), && (3.4)\\
\nabla_\omega \log \pi_\Theta(a, x|s) &= \nabla_\omega \log \pi_\omega^d(a|s)\pi_\theta^a(x|s) && (3.5)\\
&= \nabla_\omega \log \pi_\omega^d(a|s), && (3.6)\\
\nabla_\Theta \log \pi_\Theta(a, x|s) &= \begin{pmatrix} \nabla_\theta \log \pi_\theta^a(x|s) \\ \nabla_\omega \log \pi_\omega^d(a|s) \end{pmatrix}. && (3.7)
\end{aligned}$$

3.2 Related Work

A parameterized task is a problem which is defined by a set of task parameters which can specify a goal or configuration. These task parameters τ are given at the beginning of each episode, and are fixed

throughout the episode. The goal is to maximize

$$\int P(\tau)J(\pi_\tau, \tau)d\tau$$

where π_τ is a task dependent policy, J is the objective function, and $P(\tau)$ is the probability of task parameters τ.

Kober *et al.* [2012] developed algorithms to adjust dynamic motor primitives to different task parameters. They apply this to learn table-tennis and darts with different starting positions and targets. da Silva *et al.* [2012] introduced parameterized skills as task dependent parameterized policies, and devised a method for learning different skills as distinct charts in policy space. The system works by sampling a set of tasks, learning their associated parameters, and trying to find a mapping from tasks to parameters [da Silva *et al.* 2012].

Either of these methods could be used to create parameterized actions. For example in the robot soccer domain, we could train a kick action with task parameters such as the target and power and then learn another skill for running at a particular speed. The task parameters for these skills become the parameters for our parameterized actions, and we are faced with the problem of selecting which parameterized actions to use, and how to select its parameters.

Guestrin *et al.* [2004] introduced an algorithm for solving factored MDPs with a hybrid discrete-continuous action space. A hybrid discrete-continuous action is an action with both a discrete and a continuous parts, but no direct connection between the discrete and continuous spaces. Their formalism has an action space with a mixed set of discrete and continuous components, whereas our domain has distinct actions with a different number of continuous components for each action. Furthermore, they assume the domain has a compact factored representation, and only consider planning.

Rachelson [2009] encountered parameterized actions in the form of an action to wait for a given period of time in his research on time dependent, continuous time MDPs (TMDPs). He developed XMDPs, which are TMDPs with a parameterized action space [Rachelson 2009]. He developed a Bellman operator for this domain, and in a later paper mentions that the TiMDP$_{poly}$ algorithm can work with parameterized actions, although this specifically refers to the parameterized wait action [Rachelson *et al.* 2009]. This research also takes a planning perspective, and only considers a time dependent domain. Additionally, the size of the parameter space for the parameterized actions is the same for all actions. There are some similarities to parameterized actions, but in a different formalism and problem domain.

Hoey *et al.* [2013] considered mixed discrete-continuous actions in their work on Bayesian affect control theory. They model affect control theory, a formalization of interpersonal interaction, as a POMDP with hybrid actions. To approach this problem they use a form of POMCP, a Monte Carlo sampling algorithm, using domain specific adjustments to compute the continuous action components [Silver and Veness 2010]. They note that the discrete and continuous components of the action space reflect different control aspects: the discrete control provides the "what", while the continuous control describes the "how" [Hoey *et al.* 2013]. The main similarity between this work and our own is this idea, but the setting and problem is unrelated.

In their research on symbolic dynamic programming (SDP) algorithms, Zamani *et al.* [2012] considered domains with a set of discrete parameterized actions. Each of these actions has a different parameter space. Symbolic dynamic programming is a form of planning for relational or first-order MDPs, where the MDP has a set of logical relationships defining its dynamics and reward function. Their algorithms represent the value function as an extended algebraic decision diagram (XADD). As such, this work is limited to MDPs with predefined logical relations.

An option is a closed-loop policy that can be followed for multiple steps [Sutton *et al.* 1999]. Options generally represent a high-level skill. We could view each parameterized action as an option with a continuous policy. This would be useful if we want to allow for parameterized actions that are extended over a number of steps. Options are related to parameterized actions in that they are both different ways of representing skills. They differ as options do not have parameters, but can represent skills over an extended timescale. There is also a shortage of work relating to parameterized options.

3.3 Summary

Parameterized actions are actions with both a discrete and a continuous component. We have described parameterized action MDPs (PAMDPs), and explained how a policy might be designed for such a domain. Such a policy consists of a discrete policy (the action policy) and a continuous policy (the parameter policy). We have discussed two naive approaches for learning in PAMDPs: discretization and direct policy search. In the next chapter we introduce the Q-PAMDP algorithm, which alternates learning the discrete and continuous policies.

Chapter 4

The Q-PAMDP Algorithm

We want to learn a policy for a parameterized action MDP. We propose the Q-PAMDP algorithm which alternates updating the parameter-policy and learning an action-value function. We can approach learning the action-value function by fixing the parameter-policy and treating it as a discrete-action MDP. Then we can update the parameter-policy by fixing the action-value function and treating it as an optimization problem. The advantage of this approach is we can apply an appropriate method on each subproblem, rather than using one method which may performance poorly on the problem as a whole.

To fix the parameter-policy, we define the corresponding discrete-action MDP as follows. For any PAMDP $M = \langle S, A_{d,x}, P, R, \gamma \rangle$ with a fixed parameter-policy π_θ, there exists a corresponding discrete action MDP, M_θ. Definitions used in this paper are given in appendix A.

Definition 4.1 (M_θ)**.** *M_θ is a tuple given by*

$$M_\theta = \langle S, A_d, P_\theta, R_\theta, \gamma \rangle$$

where A_d is the discrete action set and

$$P_\theta(s, a, s') = \int_{x \in X_a} \pi^a_\theta(x|s) P(s, (a, x), s') dx \tag{4.1}$$

$$R_\theta(s, a, r) = \int_{x \in X_a} \pi^a_\theta(x|s) R(s, (a, x), r) dr. \tag{4.2}$$

M_θ is a discrete action MDP with a fixed parameter-policy π_θ.

The objective function for M_θ is the same as M for a fixed θ. We represent the action-value function for M_θ using function approximation with parameters ω. For M_θ, there exists an optimal set of representation weights ω^*_θ which maximizes $J_M(\theta, \omega)$ with respect to ω.

Let $W_M(\theta) = \omega^*_\theta$ be a function which selects an optimal ω for θ. We can compute $W_M(\theta)$ using an algorithm such as gradient-descent SARSA(λ) [Sutton and Barto 1998].

Definition 4.2 ($W_M(\theta)$)**.** *The function W_M is given by*

$$W_M(\theta) = \arg\max_\omega J_M(\theta, \omega).$$

Definition 4.3 ($H_M(\theta)$)**.** *The function H_M is given by*

$$H_M(\theta) = J_M(\theta, W_M(\theta)).$$

Algorithm 4 Q-PAMDP(k)

Input:
Initial parameters θ_0, ω_0
Parameter update method P-UPDATE
Q-learning algorithm Q-LEARN
Algorithm:
$\omega \leftarrow$ Q-LEARN$^{(\infty)}(M_\theta, \omega_0)$
repeat
  $\theta \leftarrow$ P-UPDATE$^{(k)}(J_\omega, \theta)$
  $\omega \leftarrow$ Q-LEARN$^{(\infty)}(M_\theta, \omega)$
until θ converges

Algorithm 4 describes a method for alternating updating θ and ω. P-UPDATE should optimize θ with respect to $H_M(\theta) = J_M(\theta, W_M(\theta)) = J_M(\theta, \omega)$. The algorithm takes two methods, P-UPDATE and Q-LEARN. Q-LEARN can be any algorithm for Q-learning with function approximation, provided Q-LEARN$(M_\theta, \omega) = W_M(\theta)$, and is left as a design choice. P-UPDATE performs a single update of θ, after which ω is learned for the new MDP M_θ. With Q-PAMDP(k), k represents the number of iterations of P-UPDATE before performing another relearning Q.

We consider two potential values for k: $k = 1$ and $k = \infty$, as these values are useful in developing our theoretical results. Q-PAMDP(1) alternates quickly, following each update of θ with multiple updates of ω. This method works better on methods where the effect of the parameters θ is continuous on the appropriate ω. We expand on this idea in the next section. Q-PAMDP(∞) represents alternating optimization: each method continues until convergence. This has the advantage of being simpler to implement and tune, and relies less on the relationship between θ and the optimal ω.

4.1 Convergence Properties of Q-PAMDP(1)

We now show that Q-PAMDP(1) converges to a local or global maximum with weak assumptions. We assume that iterating P-UPDATE converges to some θ^* with respect to a given objective function J. As the P-UPDATE step is a design choice, it can be any algorithm with the appropriate convergence property such as eNAC. Q-PAMDP(1) is equivalent to the sequence

$$\omega_{t+1} = W_M(\theta_t) \tag{4.3}$$
$$\theta_{t+1} = \text{P-UPDATE}(J_{\omega_{t+1}}, \theta_t), \tag{4.4}$$
$$\tag{4.5}$$

if Q-LEARN converges to $W_M(\theta)$ for each given θ.

Theorem 4.1.1 (Convergence to a Local Maximum)**.** *For any θ_0, if the sequence*

$$\theta_{t+1} = \textit{P-UPDATE}(H_M, \theta_t), \tag{4.6}$$

converges to a local maximum with respect to H_M, then Q-PAMDP(1) converges to a local maximum with respect to J.

Proof. By definition of the sequence above $\omega_t = W_M(\theta_t)$, so it follows that

$$J_{\omega_t} = J_M(\theta, W_M(\theta)) \tag{4.7}$$
$$= H_M(\theta). \tag{4.8}$$

In other words, the objective function J_M equals H_M if $\omega = W_M(\theta)$. If we replace J_M with H_M in our update for θ, to obtain the update rule

$$\theta_{t+1} = \text{P-UPDATE}(H_M, \theta_t). \tag{4.9}$$

Therefore by equation 4.6 the sequence θ_t converges to a local maximum (definition A.5) θ^* with respect to H_M. Let $\omega^* = W_M(\theta^*)$. As θ^* is a local maximum with respect to H_M, by definition A.5 there exists $\epsilon > 0$, $s.t.$

$$||\theta^* - \theta||_2 < \epsilon \implies H_M(\theta) \leq H_M(\theta^*). \tag{4.10}$$

Therefore for any ω,

$$\left|\left|\begin{pmatrix}\theta^* \\ \omega^*\end{pmatrix} - \begin{pmatrix}\theta \\ \omega\end{pmatrix}\right|\right|_2 < \epsilon \implies ||\theta^* - \theta||_2 < \epsilon \tag{4.11}$$

$$\implies H_M(\theta) \leq H_M(\theta^*) \tag{4.12}$$

$$\implies J_M(\theta, \omega) \leq J_M(\theta^*, \omega^*). \tag{4.13}$$

Therefore (θ^*, ω^*) is a local maximum with respect to J_M. □

In summary, if we can locally optimize θ and $\omega = W_M(\theta)$ at each step, then we will find a local maximum for $J_M(\theta, \omega)$. The conditions for the previous theorem can be met by assuming that P-UPDATE is a local optimization method such as a gradient based policy search. A similar argument shows that if the sequence θ_t converges to a global maximum with respect to H_M, then Q-PAMDP(1) converges to a global maximum (θ^*, ω^*).

One problem is that at each step we must re-learn $W_M(\theta)$ for the updated value of θ. We now show that if updates to θ are bounded and $W_M(\theta)$ is a continuous function, then the required updates to ω will also be bounded. Intuitively, we are supposing that a small update to θ results in a small change in the weights specifying which discrete action to choose. The assumption that $W_M(\theta)$ is continuous is strong, and may not be satisfied by all PAMDPs. It is not necessary for the operation of Q-PAMDP(1), but when it is satisfied we do not need to completely re-learn ω after each update to θ. We show that by selecting an appropriate learning rate α we can shrink the differences in ω as desired.

Theorem 4.1.2 (Bounded Updates to ω)**.** *If W_M is continuous with respect to θ, and updates to θ are of the form*

$$\theta_{t+1} = \theta_t + \alpha_t \textit{P-UPDATE}(\theta_t, \omega_t), \tag{4.14}$$

where α_t is the update rate, and with the norm of each P-UPDATE bounded by

$$0 < ||\textit{P-UPDATE}(\theta_t, \omega_t)||_2 < \delta,$$

for some $\delta > 0$, then for any difference in ω, for $\epsilon > 0$, there is an initial update rate $\alpha_0 > 0$ such that

$$\alpha_t < \alpha_0 \implies ||\omega_{t+1} - \omega_t||_2 < \epsilon.$$

Proof. Let $\epsilon > 0$ and

$$\alpha_0 = \frac{\delta}{||\text{P-UPDATE}(\theta_t, \omega_t)||_2}.$$

As $\alpha_t < \alpha_0$, it follows that

$$\begin{aligned}\delta &> \alpha_t \left|\left|\text{P-UPDATE}(\theta_t, \omega_t)\right|\right|_2 && (4.15)\\ &= \left|\left|\alpha_t\text{P-UPDATE}(\theta_t, \omega_t)\right|\right|_2 && (4.16)\\ &= \left|\left|\theta_{t+1} - \theta_t\right|\right|_2. && (4.17)\end{aligned}$$

So we have

$$||\theta_{t+1} - \theta_t||_2 < \delta.$$

As W_M is continuous, this means that

$$\begin{aligned}||W_M(\theta_{t+1}) - W_M(\theta_t)||_2 &= ||\omega_{t+1} - \omega_t||_2 && (4.18)\\ &< \epsilon. && (4.19)\end{aligned}$$

□

In other words, if our update to θ is bounded and W_M is continuous, we can always adjust the learning rate α so that the difference between ω_t and ω_{t+1} is bounded.

4.2 Gradient of $H_M(\theta)$

With Q-PAMDP(1) we want P-UPDATE to optimize $H_M(\theta)$. One logical choice would be to use a gradient update. The next theorem shows that gradient of H is equal to the gradient of J if $\omega = W_M(\theta)$. This is useful as we can apply existing gradient-based policy search methods to compute the gradient of J with respect to θ. The proof follows from the fact that we are at a global maximum of J with respect to ω, and so the gradient $\nabla_\omega J$ is zero. This theorem requires that W is differentiable (and therefore also continuous).

Theorem 4.2.1 (Gradient of $H_M(\theta)$)**.** *If $J_M(\theta, \omega)$ is differentiable with respect to θ and ω and $W_M(\theta)$ is differentiable with respect to θ, then the gradient of H_M is given by $\nabla_\theta H_M(\theta) = \nabla_\theta J_M(\theta, \omega^*)$, where $\omega^* = W_M(\theta)$.*

Proof. If $\theta \in \mathbb{R}^n$ and $\omega \in \mathbb{R}^m$, then we can compute the gradient of H_M by the chain rule:

$$\begin{aligned}\frac{\partial H_M(\theta)}{\partial \theta_i} &= \frac{\partial J_M(\theta, W_M(\theta))}{\partial \theta_i} && (4.20)\\ &= \sum_{j=1}^{n} \frac{\partial J(\theta, \omega^*)}{\partial \theta_j}\frac{\partial \theta_j}{\partial \theta_i} + \sum_{k=1}^{m} \frac{\partial J(\theta, \omega^*)}{\partial \omega_k^*}\frac{\partial \omega_k^*}{\partial \theta_i} && (4.21)\\ &= \frac{\partial J_M(\theta, \omega^*)}{\partial \theta_i} + \sum_{k=1}^{m} \frac{\partial J(\theta, \omega^*)}{\partial \omega_k^*}\frac{\partial \omega_k^*}{\partial \theta_i}, && (4.22)\end{aligned}$$

where $\omega^* = W_M(\theta)$. Note that by definitions of W_M,

$$\begin{aligned}\omega^* &= W_M(\theta) && (4.23)\\ &= \arg\max_\omega J_M(\theta, \omega), && (4.24)\end{aligned}$$

we have that the gradient of J_M with respect to ω is zero

$$\frac{\partial J_M(\theta, \omega^*)}{\partial \omega_k^*} = 0,$$

as ω is a global maximum with respect to J_M for fixed θ when $\omega = \omega^*$. It follows that

$$\nabla_\theta H_M(\theta) = \nabla_\theta J_M(\theta, \omega^*).$$

□

To summarize, if $W_M(\theta)$ is continuous and P-UPDATE converges to a global or local maximum, then Q-PAMDP(1) will converge to a global or local maximum respectively, and the Q-LEARN step will be bounded if the update rate of the P-UPDATE step is bounded. As such, if P-UPDATE is a policy gradient update step then Q-PAMDP by Theorem 4.3 will converge to a local maximum and by Theorem 4.4 the Q-LEARN step will require a fixed number of updates. This policy gradient step can use the gradient of J with respect to θ.

4.3 Convergence properties of Q-PAMDP(∞)

With Q-PAMDP(∞) each step performs a full optimization on θ and then a full optimization of ω. The θ step would optimize $J_M(\theta, \omega)$, not $H_M(\theta)$, as we do update ω while we update θ. Q-PAMDP(∞) has the disadvantage of requiring global convergence properties for the P-UPDATE method.

Theorem 4.3.1 (Local Convergence of Q-PAMDP(∞))**.** *If at each step of Q-PAMDP(∞) for some bounded set Θ:*

$$\theta_{t+1} = \arg\max_{\theta \in \Theta} J_M(\theta, \omega_t), \tag{4.25}$$

$$\omega_{t+1} = W_M(\theta_{t+1}), \tag{4.26}$$

then Q-PAMDP(∞) converges to a local maximum.

Proof. By definition of W_M, $\omega_{t+1} = \arg\max_\omega J_M(\theta_{t+1}, \omega)$. This algorithm takes the form of direct alternating optimization. As such, it converges to a local maximum[Bezdek and Hathaway 2002]. □

Q-PAMDP(∞) has weaker convergence properties than Q-PAMDP(1), as it requires a globally convergent P-UPDATE. However, it has the potential to bypass nearby local maxima [Bezdek and Hathaway 2002]. Note that Q-PAMDP(1) and Q-PAMDP(∞) will generally converge to a local maximum and which local maximum they converge to depends on the initial value of θ.

4.4 Summary

We have introduced the Q-PAMDP(k) algorithm for model-free learning in PAMDPs. Q-PAMDP(k) alternates learning a parameter and an action policy. Two variances of Q-PAMDP(k): Q-PAMDP(1) and Q-PAMDP(∞). These algorithms differ by how they frequently they alternate learning different policies. We considered the convergence properties of Q-PAMDP(1) and Q-PAMDP(∞). By making weak assumptions both methods convergence to a local maximum. Q-PAMDP(1) requires a locally convergent P-UPDATE method and Q-PAMDP has the stronger requirement that the P-UPDATE method be globally convergent. In the next two chapters we compare these algorithms against direct optimization and SARSA in two domains: the Goal domain and the Platform domain.

Chapter 5

The Robot Soccer Goal Domain

Robot soccer is a challenge domain for designing autonomous soccer-playing robots. The difficulty of this problem allows for the exploration of a number of different problems in artificial intelligence such as planning, co-operation, skill-development, vision, communication, and state inference. We consider a two-dimensional simulated robot soccer problem, depicted in Figure 5.1.

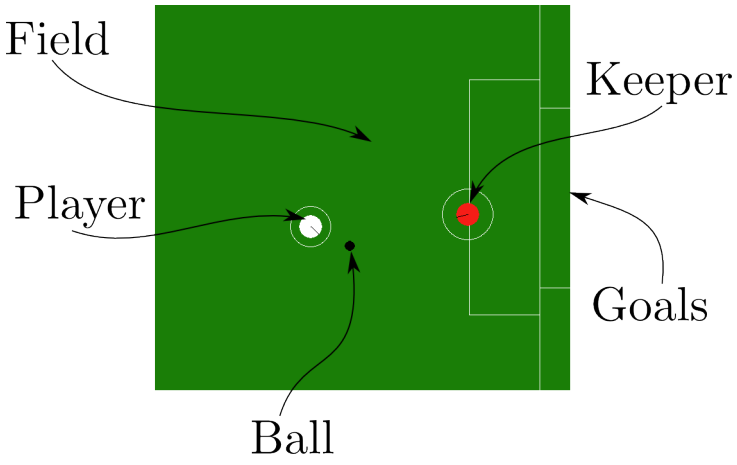


Figure 5.1: The robot soccer 2D simulator.

Each robot is represented by a circular entity with a position (x, y), velocity $(\dot{x}, \dot{y})$, and an orientation θ. There are three primitive actions:

- turn(θ): turns the player θ degrees.
- dash(ρ): accelerates the player forward with power ρ.
- kick(ρ, θ): if the ball is in the player's possession, accelerates the ball with power ρ in the direction θ.

Each action adds some noise to the control. The simulator uses a simple physics system: if two entities overlap they are shifted apart, and the positions are updated with the rules:

$$\hat{v}_{t+1} = d\hat{v}_t + \hat{a}_{t+1} \tag{5.1}$$

$$\hat{p}_{t+1} = \hat{p}_t + \hat{v}_{t+1}, \tag{5.2}$$

for position $\hat{p}$, velocity $\hat{v}$, and acceleration $\hat{a}$ and where d is the velocity decay factor. The decay factor represents how much the velocity slows down at each step. We describe the implementation details of this domain in appendix D.

In the robot soccer goal domain, a single agent (the player) attempts to score a goal against against an adversary (the keeper). The keeper is a fixed, predefined agent. This domain is depicted in figure 5.2. Our goal is develop a policy for the player that can score goals against the keeper.

5.1 Domain

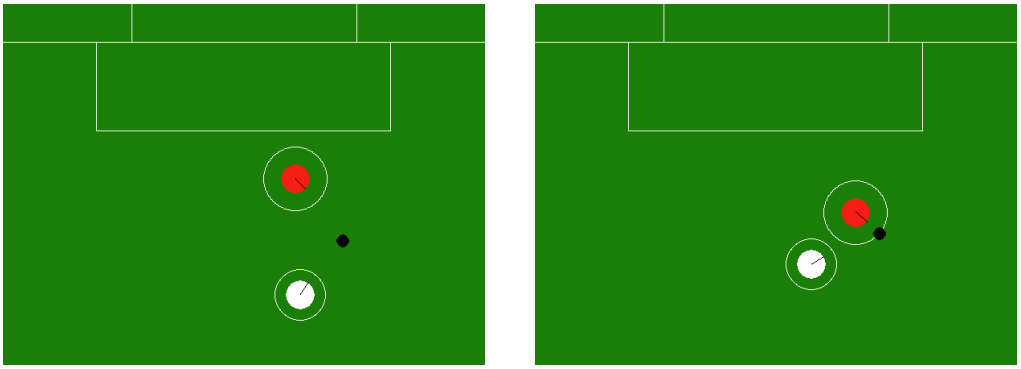


Figure 5.2: The robot soccer goal domain. The player (white) attempts to kick the ball (black) past the keeper (red) and into the goal. The radius around players depicts their area of interaction with the ball.

The state space is defined by the following variables:

$$p_x, p_y, \dot{p}_x, \dot{p}_y, p_\theta,$$

the player's position, velocity, and orientation,

$$g_x, g_y, \dot{g}_x, \dot{g}_y, g_\theta$$

the keeper's position, velocity, and orientation, and

$$b_x, b_y, \dot{b}_x, \dot{b}_y$$

the ball's position and velocity. The state space therefore has 14 state variables.

Although the primitive actions turn, dash, and kick represent all the variety of actions we require to play the game, we wish to use higher-level actions. We hand-designed the skills *to-ball*, *kick-to*, and *shoot-goal*.

The *to-ball*() skill is used to move the player towards the ball until it gains possession of the ball. If the player is facing the ball, the player simply dashes at power 10, otherwise the player turns to face the ball.

The *kick-to*(x, y) is a skill that kicks the ball towards position $\hat{p} = (x, y)$. This requires computing the required power and direction for such a kick. First, note that the ball starts at position $\hat{b}_0$ and velocity $\hat{v}_0$, and if we assume no other force acts on it, then the system is given by

$$\hat{v}_{t+1} = d\hat{v}_t \tag{5.3}$$

$$\hat{b}_{t+1} = \hat{b}_t + \hat{v}_{t+1}. \tag{5.4}$$

Expanding this equation, we have that

$$\hat{b}_t = \hat{b}_0 + \sum_{i=0}^{t-1} d^i \hat{v}_0 \tag{5.5}$$

$$= \hat{b}_0 + \frac{1-d^t}{1-d}\hat{v}_0. \tag{5.6}$$

Note that the velocity diminishes exponentially, but is always positive provided $\hat{v_0}$ is non-zero. As such, to kick the ball to a position p, we must have that

$$\hat{p} = \hat{b}_\infty \tag{5.7}$$

$$= \lim_{t\to\infty} \hat{b}_t \tag{5.8}$$

$$= \lim_{t\to\infty} \hat{b}_0 + \frac{1-d^t}{1-d}\hat{v}_0 \tag{5.9}$$

$$= \hat{b}_0 + \frac{1}{1-d}\hat{v}_0. \tag{5.10}$$

Therefore, if we want to kick the ball to position (x, y), it must start at velocity $\hat{v}_0$, and then

$$\hat{v}_0 = (1-d)(\hat{p} - \hat{b}_0).$$

If the ball is already moving at velocity $\hat{v}$ we must then impart an acceleration $\hat{a}$ so that

$$\hat{a} + \hat{v} = \hat{v}_0.$$

Therefore

$$\hat{a} = \hat{v}_0 - \hat{v} \tag{5.11}$$

$$= (1-d)(\hat{p} - \hat{b}_0) - \hat{v}. \tag{5.12}$$

The *kick-to* command is reduced to kick(ρ, θ), where

$$\rho = ||\hat{a}||_2 \tag{5.13}$$

$$\theta = \arctan 2(\hat{a}_1, \hat{a}_0). \tag{5.14}$$

The *shoot-goal*(h) action kicks the ball into the net at position h along the goal line. This is simply reduced to a *kick-to* action with target (GOAL-LINE + BALL-WIDTH, h). We kick to a target inside the net by one BALL-WIDTH.

Each episode starts with the player at a random position along the left bound of the field. The player starts with the ball in its possession, and the keeper is positioned between the ball and the goal. The game takes place in a 2D environment where the player and the keeper have a position, velocity and orientation and the ball has a position and velocity resulting in 14 continuous state variables.

An episode ends when the keeper possesses the ball, the player scores a goal, or the ball leaves the field. The reward for an action is 0 for non-terminal state, 50 for a terminal goal state, and $-\kappa$ for a terminal non-goal state, where κ is the distance of the ball to the goal. The player has two parameterized actions: *kick-to*(x, y), and *shoot-goal*(h). If the player is not in possession of the ball, it moves towards it using *to-ball*(). The keeper has a fixed policy: it moves towards the ball, and if the player shoots at the goal, the keeper moves to intercept the ball.

To score a goal, the player must shoot around the keeper. This means that at some positions we must shoot left past the keeper, and at others shoot to the right past the keeper. However at no point do we shoot at the keeper, so an optimal policy would be discontinuous. This policy would be difficulty to represent in a purely continuous action space, but is simpler in a parameterized action domain. We split the action into two parameterized actions: *shoot-goal-left*(h), *shoot-goal-right*(h). This allows us to use a simple action selection policy instead of complex continuous action policy.

5.2 Implementation Details

We represent the action-value function for the discrete action a using linear function approximation: $Q_\omega(s,a) = \omega_a^T \phi_a(s)$, where ω_a is a vector of weights, and $\phi_a(s)$ gives the features for state s. For this domain, we use Fourier basis features [Konidaris *et al.* 2011]. As we have 14 state variables, we must be selective in which basis functions to use. We only use basis functions with two non-zero elements and exclude all velocity state variables. We use the soft-max discrete action policy [Sutton and Barto 1998]

$$\pi_\omega^d(a|s) = \frac{\exp(Q_\omega(s,a)/\tau)}{\sum_{b \in A_d} \exp(Q_\omega(s,b)/\tau)},$$

where τ is the action selection temperature. For the action-policy, we use a Fourier basis of degree 7, which results in 1057 basis functions. We use three parameterized actions: kick-to, shoot-goal-left, shoot-goal-right. Therefore ω contains 3171 variables.

The features and initial values for θ were specifically designed to make the initial parameter-policy useful. We represent the action-parameter policy π_θ^a as a normal distribution around a weighted sum of features

$$\pi_\theta^a(x|s) = \mathcal{N}(\theta_a^T \psi_a(s), \Sigma),$$

where θ_a is a matrix of weights, and $\psi_a(s)$ gives the features for state s, and Σ is a fixed covariance matrix. We use specialized features for each action. For the *shoot-goal* actions we use using a simple linear basis

$$\psi_{\text{sg}} = \begin{bmatrix} 1 \\ g \end{bmatrix},$$

where g is the projection of the keeper onto the goal line. For *kick-to* we use linear features

$$\psi_{\text{kt}}(s) = \left[1, bx, by, bx^2, by^2, \frac{bx - kx}{||b-k||_2}, \frac{by - ky}{||b-k||_2}\right]^T,$$

where (bx, by) is the position of the ball and (kx, ky) is the position of the keeper. These linear features allow for a policy that directs the ball around the keeper and towards the goal.

The policy is differentiable, allowing us to use a policy gradient method. To represent the action-value function we use a polynomial basis which considers only the position variables. This is because using the Fourier basis described previously would require too many parameters to optimize. For the direct policy search approach, we use the episodic natural actor critic (eNAC) algorithm [Peters and Schaal 2008], optimizing $J_M(\omega, \theta)$ with respect to (ω, θ). This approach represents the performance of currently available methods.

For the Q-PAMDP(1) and Q-PAMDP(∞) approaches we use the gradient-descent SARSA(λ) algorithm for Q-learning, and the eNAC algorithm for policy search [Peters and Schaal 2008]. Each eNAC update uses 50 episodes to estimate the gradient of J with respect to θ. For the direct policy search approach, we

use the episodic natural actor critic (eNAC) algorithm [Peters and Schaal 2008], computing the gradient of $J_M(\omega, \theta)$ with respect to (ω, θ). For the Q-PAMDP approach we use the gradient-descent SARSA(λ) algorithm for Q-learning, and the eNAC algorithm for policy search. At each step we perform one eNAC update based on 50 episodes and then refit Q_ω using 50 gradient descent SARSA(λ) episodes.

5.3 Results

We plot return in figure 5.3. As it is easier to interpret, we plot goal scoring probability in figure 5.4. The goal scoring probability refers to the probability of scoring a goal in a given episode. Return is directly correlated with goal scoring probability, so their graphs are close to indentical. We can see that direct eNAC is outperformed by Q-PAMDP(1) and Q-PAMDP(∞). This is likely due to the difficulty of optimizing the action selection parameters directly, rather than with Q-learning.

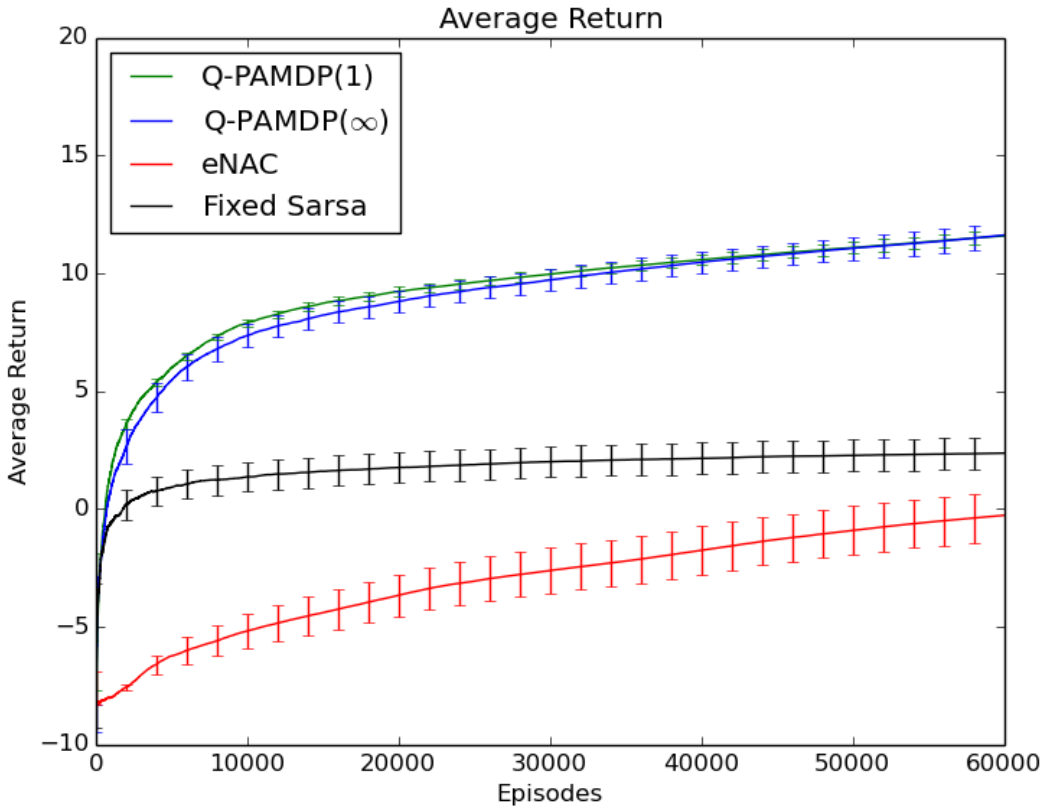


Figure 5.3: Average return, averaged over 20 runs for Q-PAMDP(1), Q-PAMDP(∞), fixed parameter SARSA, and eNAC in the goal domain. Error bars show standard error.

While the initial policy rarely scores a goal, both Q-PAMDP(1) and Q-PAMDP(∞) increase the probability of a goal to roughly 35%. Direct eNAC converged to a local maximum of 15%. Finally, we include the performance of SARSA(λ) where the action parameters are fixed at the initial θ_0. This achieves roughly 20% scoring probability. Figure 5.5 depicts a single episode using a converged Q-PAMDP(1) policy— the player draws the keeper out and strikes when the goal is open.

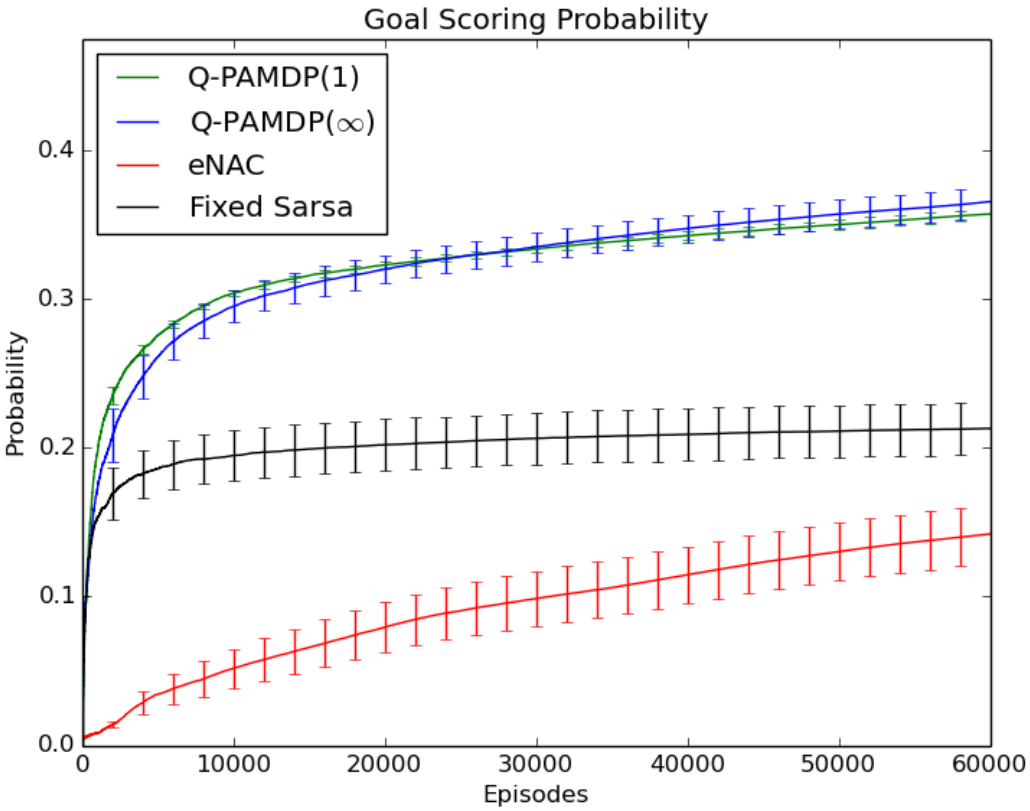


Figure 5.4: Average goal scoring probability, averaged over 20 runs for Q-PAMDP(1), Q-PAMDP(∞), fixed parameter SARSA, and eNAC in the goal domain. Intervals show standard error.

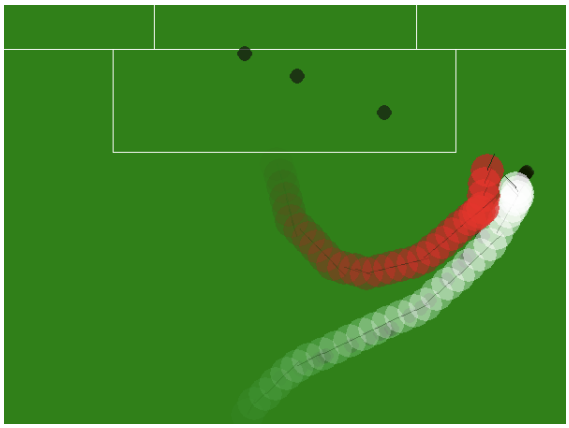


Figure 5.5: A robot soccer goal episode using a converged Q-PAMDP(1) policy. The player runs to one side, then shoots immediately upon overtaking the keeper.

Chapter 6

The Platform Domain

Next we consider the Platform domain, where the agent starts on a platform and must reach a goal while avoiding enemies. If the agent reaches the goal platform, touches an enemy, or falls into a gap between platforms, the episode ends. This domain is depicted in figure 6.1.

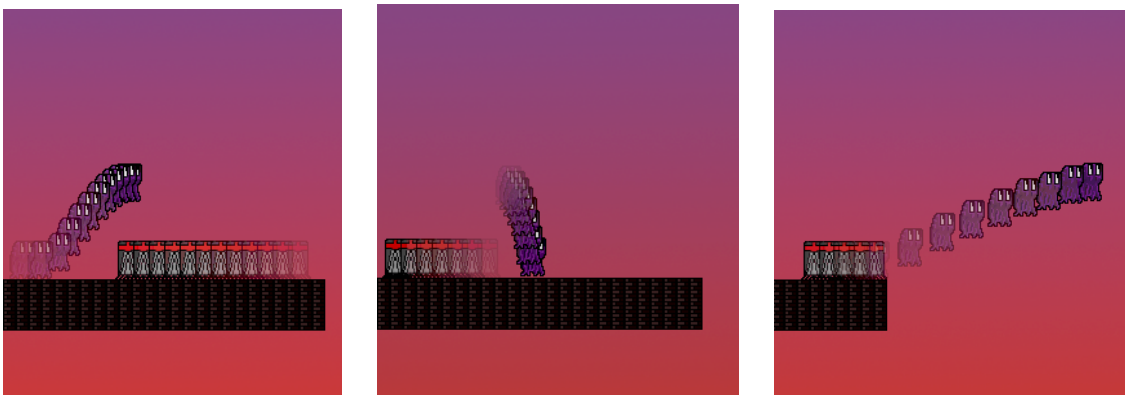


Figure 6.1: A screenshot from the Platform domain. The player (purple) hops over an enemy (grey and red), and then leaps over a gap.

6.1 Specification

The Platform domain has three types of objects: players, enemies, and platforms. The domain is 2-dimensional, and we specify positions by two variables x, y. Platforms are stationary and can be specified by a position and a width. Players and enemies have positions as well as velocities which are denoted $\dot{x}, \dot{y}$. The components of the domain are depicted in figure 6.2.

Each enemy is placed on a platform and "patrols" the platform, moving from one end to the other at a constant speed. An enemy only moves when the player is on or above its platform, and we can't move backwards, so we only need to consider one enemy at the time. As an enemy's y is fixed and its velocity $\dot{y} = 0$, we can specify an enemy's position by its position x and velocity $\dot{x}$. The agent can only take an action when on a platform, so we can assume $y = 0$ and $\dot{y} = 0$. The state space is given by the following 5 variables:

$$p_x, \dot{p}_x, e_x, \dot{e}_x, \dot{e}_x.$$

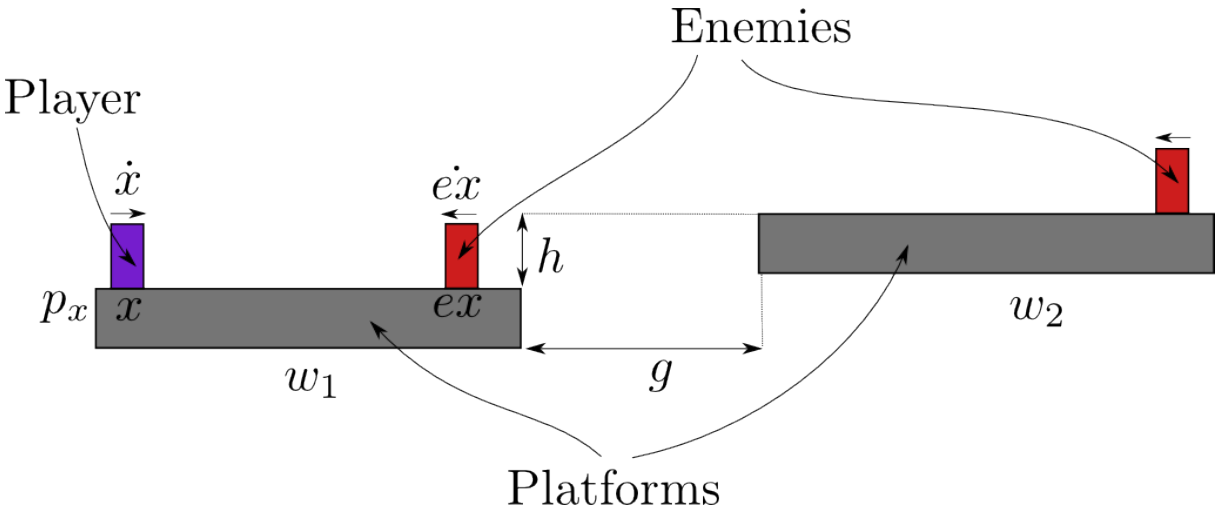


Figure 6.2: The components of the Platform domain. x, $\dot{x}$, ex, and $\dot{ex}$ represent the positions and velocities of the player and an enemy respectively. p_x is the position of the first platform, w_1 and w_2 are the width of the platforms, while g and h are the horizontal and vertical gaps between platforms.

The reward for a step is the change in x value for that step, divided by the total length of all the platforms and gaps. The agent has two primitive actions: run or jump, which continue for a fixed period or until the agent lands again respectively. The *run* action accelerates the agent forward. There are two different kinds of jumps: a high jump to get over enemies (*hop*), and a long jump (*leap*) to get over gaps between platforms. The domain therefore has three parameterized actions: *run*(dx), *hop*(dx), and *leap*(dx) where dx is the speed applied for each action. These actions are depicted in figure 6.3. The agent only takes actions while on the ground. The source code for this domain is available at `http://Github.com/WarwickMasson/aaai-platformer`.

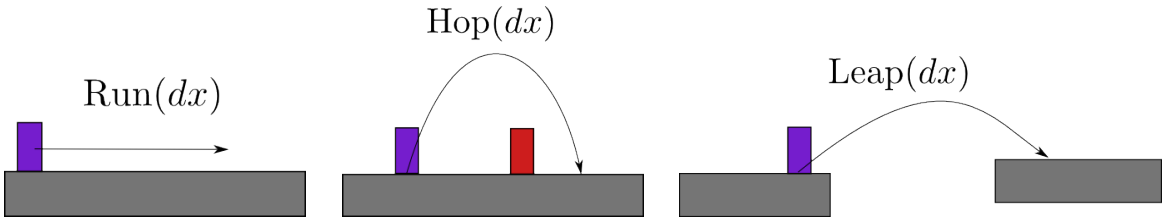


Figure 6.3: The Platform domain's actions: *run*, *hop*, and *leap*.

6.2 Approach

The state space consists of four variables $(x, \dot{x}, ex, \dot{ex})$, representing the agent position, agent speed, enemy position, and enemy speed respectively. For learning Q_ω, as in the previous domain, we use linear function approximation with the Fourier basis of degree 6. To reduce the number of basis functions, we do not use the full degree of coupling between state variables. This results in 171 basis functions, and therefore ω contains 513 variables. We apply a softmax discrete action policy with Q_ω.

We use a Gaussian parameter-policy with parameter features

$$\psi_a(s) = [1, x, \dot{x}, ex, \dot{ex}, w_1, w_2, g, p_x, h]^T,$$

where w_1 is the width of the current platform, w_2 is the width of the next platform, g is the gap between platforms, p_x is the x position of the current platform and h is the difference in height between platforms. We have labeled these parameters in figure 6.2. All parameters are scaled to between $[-1, 1]$ so they have a similar effect on the parameter-policy.

6.3 Results

Figure 6.4 shows the performance of eNAC, Q-PAMDP(1), Q-PAMDP(∞), and SARSA with fixed parameters. For eNAC, Q-PAMDP(1), Q-PAMDP(∞), the gradient update was computed using 50 episodes of samples with an eNAC gradient. For each step of Q-PAMDP(1) we performed one eNAC update, then relearned Q with 10 episodes of SARSA. For Q-PAMDP(∞) each alternation used 180 updates of eNAC then relearned Q with 1000 episodes of SARSA. Both Q-PAMDP(1) and Q-PAMDP(∞) outperformed the fixed parameter SARSA method, reaching on average 50% and 65% of the total distance respectively. This is better than the fixed SARSA baseline of 40%, and much better than direct optimization using eNAC which reached 10%.

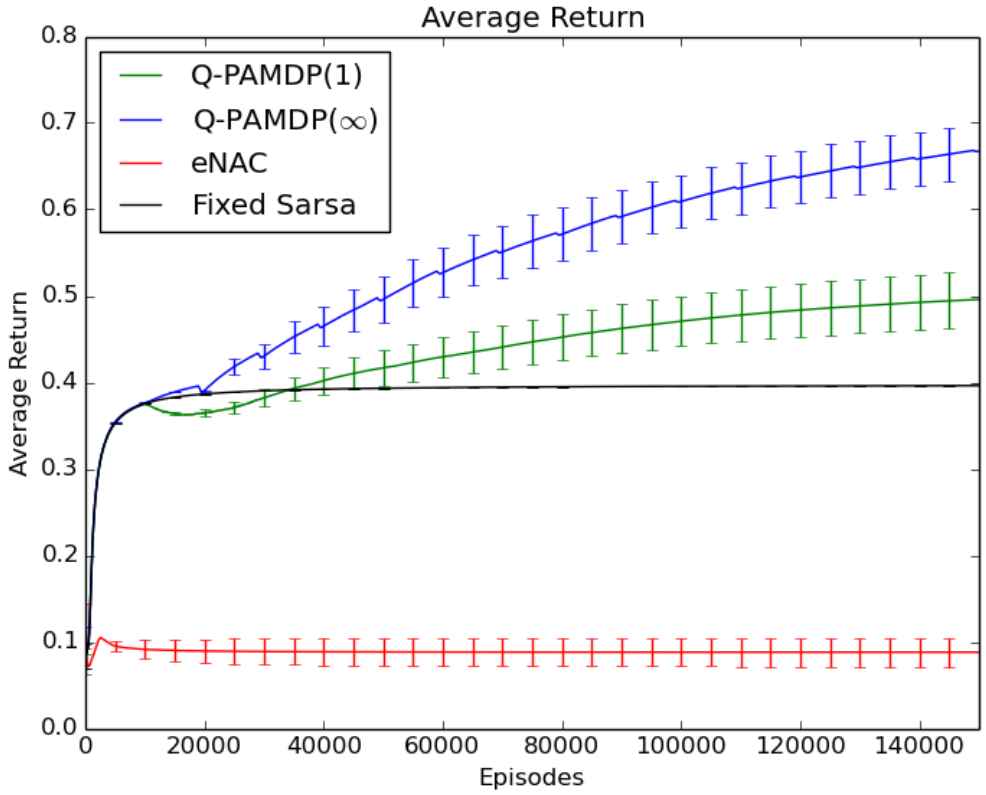


Figure 6.4: Average percentage distance covered, averaged over 20 runs for Q-PAMDP(1), Q-PAMDP(∞), and eNAC in the Platform domain. Intervals show standard error.

With Q-PAMDP(∞), there is a noticeable jagged pattern. This is likely due to the the change in performance when updating θ versus updating ω. Similarly, there is also a slight initial dip in performance for Q-PAMDP(1), which occurs after initial learning of Q_{θ_0}.

Figure 6.5 shows a successfully completed episode of the Platform domain.

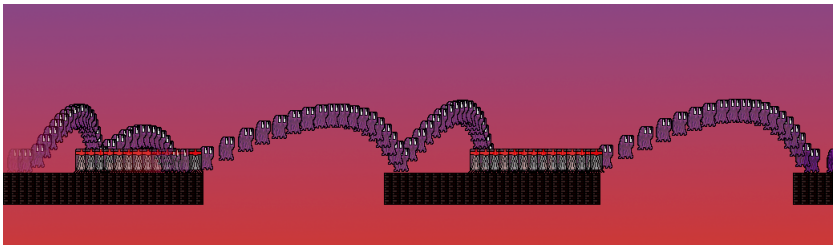


Figure 6.5: A successful episode of the Platform domain. The agent hops over the enemies, leaps over the gaps, and reaches the last platform.

6.4 Summary

The Platform domain is a simulated platforming game. We demonstrated that both versions of Q-PAMDP outperformed fixed parameter SARSA and eNAC. Overall, Q-PAMDP(∞) performed better than Q-PAMDP(1).

Chapter 7

Conclusions and Future Work

7.1 Future Work

One aspect of parameterized actions considered in this dissertation is using multiple parameterizations of a single action. This would be useful in a domain which has a complicated optimal policy or for a domain where the optimal policy is discontinuous, and therefore difficult to represent with a single parameterization.

In a high dimensional continuous action space, we can learn a number of low dimensional parameterized skills which become the parameterized actions for a domain. The work of da Silva *et al.* [2012] and of Kober *et al.* [2012] would be particularly useful for the development of such skills.

One consideration would be combining the learning of θ and ω into a single step. With an appropriate learning rate, one might be able to update θ and ω together. This would require that an approximation of $H_M(\theta)$ be made using $J_M(\theta, \omega)$ where ω is sufficiently close to $W_M(\theta)$.

Another avenue of research would be to connect the computation of $\nabla_\theta J_M(\theta, W_M(\theta))$ and knowledge of $Q(s, a)$. This might lead to an actor-critic style algorithm, where we use a TD-error computed using $Q(s, a)$ to compute the gradient [Bhatnagar *et al.* 2009].

Further work is also needed to establish the theoretical standing of Q-PAMDP(∞), and to clarify its theoretical relationship with Q-PAMDP(1). It would also be useful to determine in what domains we would expect Q-PAMDP(1) to out-perform Q-PAMDP(∞) or vice versa.

7.2 Conclusion

The PAMDP formalism models reinforcement learning domains with parameterized actions. Parameterized actions give us the adaptibility of continuous domains and the ability to use distinct kinds of actions. They also allow for simple representation of discontinuous policies without complex parameterizations. We have presented three approaches for model-free learning in PAMDPs: direct optimization and two variants of the Q-PAMDP algorithm. We have shown that Q-PAMDP(1), with an appropriate P-UPDATE method, converges to a local or global maximum. Q-PAMDP(∞) with a global optimization step converges to a local maximum.

We have examined the performance of these approaches in the goal domain and the Platformer domain. The robot soccer goal domain models the situation where a striker must out-maneuver a keeper to score

a goal. Of these, Q-PAMDP(1) and Q-PAMDP(∞) outperformed eNAC and fixed parameter SARSA. Q-PAMDP(1) and Q-PAMDP(∞) performed similarly well in terms of goal scoring, learning policies that score goals roughly 35% of the time. In the Platform domain we found that both Q-PAMDP(1) and Q-PAMDP(∞) outperformed eNAC and fixed SARSA.

By exploring the idea of reinforcement learning with parameterized actions, we can expand the range of potential applications of reinforcement learning. In particular, we can deal with more complex action spaces. The ultimate goal is to apply these advances to handle real-world control problems.

References

[Bezdek and Hathaway 2002] JC Bezdek and RJ Hathaway. Some notes on alternating optimization. In *Advances in Soft Computing*, pages 288–300. Springer, 2002.

[Bhatnagar *et al.* 2009] S Bhatnagar, RS Sutton, M Ghavamzadeh, and M Lee. Natural Actor–Critic Algorithms. *Automatica*, 45(11):2471–2482, 2009.

[da Silva *et al.* 2012] BC da Silva, G Konidaris, and AG Barto. Learning parameterized skills. In *Proceedings of the Twenty-Ninth International Conference on Machine Learning*, pages 1679–1686, 2012.

[Deisenroth *et al.* 2013] MP Deisenroth, G Neumann, and J Peters. *A Survey on Policy Search for Robotics*, volume 2. Now Publishers, 2013.

[Guestrin *et al.* 2004] C Guestrin, M Hauskrecht, and B Kveton. Solving factored MDPs with continuous and discrete variables. In *Proceedings of the Twentieth Conference on Uncertainty in Artificial Intelligence*, pages 235–242, 2004.

[Hoey *et al.* 2013] J Hoey, T Schroder, and A Alhothali. Bayesian affect control theory. In *Proceedings of the Fifth International Conference on Affective Computing and Intelligent Interaction*, pages 166–172. IEEE, 2013.

[Kober *et al.* 2012] J Kober, A Wilhelm, E Oztop, and J Peters. Reinforcement learning to adjust parametrized motor primitives to new situations. *Autonomous Robots*, 33(4):361–379, 2012.

[Konidaris *et al.* 2011] G Konidaris, S Osentoski, and PS Thomas. Value function approximation in reinforcement learning using the Fourier basis. In *Proceedings of the Twenty-Fifth International Conference on Artificial Intelligence*, pages 380–385, 2011.

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

[Rachelson *et al.* 2009] E Rachelson, P Fabiani, and F Garcia. TiMDP-poly: an improved method for solving time-dependent MDPs. In *Proceedings of the Twenty First International Conference on Tools with Artificial Intelligence*, pages 796–799. IEEE, 2009.

[Rachelson 2009] E Rachelson. *Temporal Markov Decision Problems: Formalization and Resolution*. PhD thesis, University of Toulouse, France, March 2009.

[Silver and Veness 2010] D Silver and J Veness. Monte-Carlo planning in large POMDPs. In *Advances in Neural Information Processing Systems*, volume 23, pages 2164–2172, 2010.

[Sutton and Barto 1998] RS Sutton and AG Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1998.

[Sutton *et al.* 1999] RS Sutton, D Precup, and S Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181–211, 1999.

[Zamani *et al.* 2012] Z Zamani, S Sanner, and C Fang. Symbolic dynamic programming for continuous state and action MDPs. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

Appendices

Appendix A

Theoretical Definitions

The following are the exact definitions used in this paper.

Definition A.1 (Euclidean Norm)**.** *For a vector $x \in \mathbb{R}^n$,*

$$||x||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

Definition A.2 (Continuity)**.** *A function $f(x)$, where $f : \mathbb{R}^n \to \mathbb{R}^m$ is said to be continuous with respect to x if for any $\epsilon > 0$, $\exists \delta > 0$ such that $\forall x, y \in \mathbb{R}^n$,*

$$||x - y||_2 < \delta \implies ||f(x) - f(y)||_2 < \epsilon.$$

Definition A.3 (Convergence of a Sequence)**.** *We say that a sequence $\{a_n\}$ converges to a if for any $\epsilon > 0$, $\exists N \in \mathbb{N}$ such that*

$$n \geq N \implies ||a_n - a||_2 < \epsilon.$$

Definition A.4 (Global Maximum)**.** *For a function $f : \mathbb{R}^n \to \mathbb{R}$, we say that x^* is a global maximum for f if for any $x \in \mathbb{R}^n$,*

$$f(x) \leq f(x^*).$$

Definition A.5 (Local Maximum)**.** *For a function $f : \mathbb{R}^n \to \mathbb{R}$, we say that x^* is a local maximum for f if there exists $\epsilon > 0$ such that for any x*

$$||x - x^*||_2 < \epsilon \implies f(x) \leq f(x^*).$$

Appendix B

Disconnected Action Spaces

One use of the PAMDPs formalism is for handling disconnected action spaces. By disconnected action spaces, we mean that there are sets $X_1, X_2, \ldots, X_k$, such that

$$A = \bigcup_{i=1}^{k} X_i,$$

but where for each X_i, X_j there is no path between one element of X_i to an element of X_j.

Consider a robotics problem involving a robot arm sorting balls into two boxes. At each step, the robot is given a ball of a given colour and size and must place it into the correct box. The colour of the ball determines which box it should be sorted into, while the size determines where it should be placed in the box. We have an action drop-ball-at(x, y), which drops the ball at position (x, y). We can only drop a ball into a box, any other position is invalid. The action space for this problem would look like figure B.1: two disconnected areas B_1 and B_2, with the rest of the space $\mathbb{R}^2$ an invalid target. While we could treat the space $A = B_1 \bigcup B_2$ as a single continuous space, it makes more sense to treat these as two distinct actions with specific parameters. This is where parameterized actions are helpful: we can have two parameterized actions $(a_1, (x, y))$ and $(a_2, (x, y))$ corresponding to spaces B_1 and B_2.

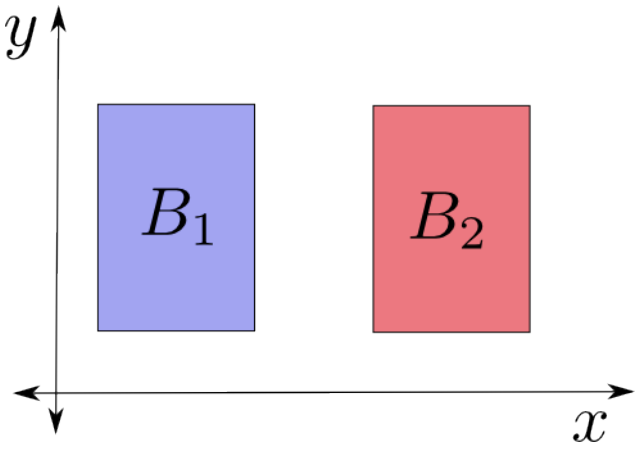


Figure B.1: The ball sorting problem: balls must be dropped into one of the boxes, but not in between them.

Appendix C

Discontinuous Policies

In continuous parameterized policies, we may encounter a situation where the optimal policy is discontinuous. As an example, consider a simple domain where we have a single state variable $x \in \mathbb{R}$, with $x_0 = 0$, where we can apply action $a_t \in \mathbb{R}$ such that $x_{t+1} = x_t + a_t$, and we must move to either $x = -b$ or $x = b$, whichever is closer. This domain is depicted in figure C.1.

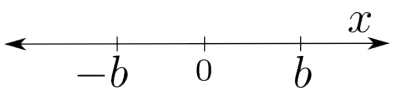


Figure C.1: Discontinuous policy domain. The agent must move to either $-b$ or b, whichever is closer.

The optimal policy for this domain is given by a piecewise continuous function

$$\pi^*(x) = \begin{cases} b - x & \text{if} \quad x \geq 0 \\ x - b & \text{if} \quad x < 0 \end{cases}.$$

This policy is depicted in figure C.2.

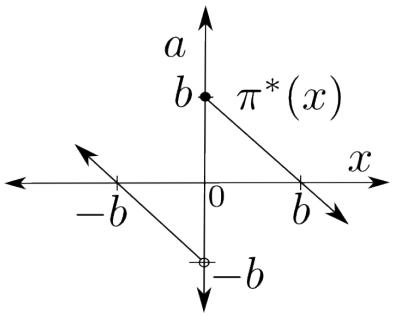


Figure C.2: Optimal policy $\pi^*(x)$ for the discontinuous policy domain.

Now consider how to represent this using a parameterized policy. A standard parameterized policy uses a linear form $\pi(s) = \theta^T \phi(s)$, where the features $\phi(s)$ tend to be linear with respect to s. Such a policy cannot represent the optimal policy π^*. This means we must specially design these features to cope with the discontinuity. If we plan to use a gradient update the policy must also be differentiable, which can complicate matters. We could alternatively treat this as a parameterized action problem: create two actions a_+ and a_-, and therefore create two policies which are linear with respect to x. This particular strategy is used in the goal domain for the *shoot-goal* action.

Appendix D

RSS Simulator

The 2D Robo-cup soccer server (RCSS) is a simplified simulated version of the robot soccer problem. Different sub-problems of soccer can be considered in the simulator. The soccer server is designed as a single server that agents connect to via sockets. It involves a complex message passing system, and systems for monitoring, player vision, and communication between players. These features are unnecessary and at times create obstacles for its use in this domain: the requirement of the server to be running independently means the server runs at its own schedule, which can complicate the use of learning agents and increase running times. More-over, the server uses message passing through UDP, which means that messages can be lost. The requirement that the messages be parsed from S-expressions adds additional computational load to the problem.

For all these reasons, we created a limited form of the simulator that implements the simulation rules. We ported these update rules to a python package. The simulator updates only when called with a new action, and as such there is no possibility of messages being lost. We implemented a visualizer for the game using the pygame package.

The source code for this domain is available at `http://github.com/WarwickMasson/aaai-goal`. This code is divided into modules for running the simulator, constructing the interface, running learning experiments, and drawing graphs.