Research Proposal

Model-Based Reinforcement Learning with Parameterised Action Spaces

Craig Bester
783135

Supervised by:

Pravesh Ranchod
George Konidaris
Steven James

School of Computer Science and Applied Mathematics
University of the Witwatersrand
May 9, 2017

Abstract

The recent formulation of parameterised action spaces for reinforcement learning has moved the field towards solving more complex domains that require learning a mix of high-level actions and low-level control to perform tasks, such as robotics. We note that current algorithms for parameterised action spaces are model-free, and require many learning episodes to converge to good policies. We propose applying model-based techniques in order to increase data efficiency and learning speed. In particular, we propose using an incremental actor-critic algorithm combined with Dyna-style planning updates for the parameter update step of Q-PAMDP. The modified model-based Q-PAMDP algorithm will be compared against the original model-free algorithm on the Platform and Robot Soccer Goal domains, with the learning speed evaluated in terms of the number of training episodes taken to converge to a good policy.

UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
School of Computer Science and Applied Mathematics

Declaration

2017

I, _____, (Student number: _____)

am a student registered for Introduction to Research Methods in 2017.

This declaration applies to the _____ document of Introduction to Research Methods.

I hereby declare the following:

- I am aware that plagiarism (the use of someone else's work without their permission and/or without acknowledging the original source) is wrong.
- I confirm that the work submitted for assessment is my own unaided work except where I have explicitly indicated otherwise.
- I have followed the required conventions in referencing the thoughts and ideas of others.
- I understand that the University of the Witwatersrand, Johannesburg may take disciplinary action against me if there is a belief that this is not my own unaided work or that I have failed to acknowledge the source of the ideas or words in my writing.

Signature: _____ Date: _____

Contents

Abstract i

Declaration ii

List of Figures iv

1 Introduction 1

2 Background and Related Work 3
- 2.1 Introduction . . . 3
- 2.2 Reinforcement Learning . . . 3
  - 2.2.1 Value Function Approximation . . . 5
  - 2.2.2 Continuous Action Spaces . . . 7
  - 2.2.3 Parameterised Action Spaces . . . 10
- 2.3 Model-Based Reinforcement Learning . . . 12
  - 2.3.1 Dyna . . . 12
- 2.4 Domains . . . 13
  - 2.4.1 Platform Domain . . . 13
  - 2.4.2 Robot Soccer Goal Domain . . . 13
- 2.5 Related Work . . . 14
- 2.6 Conclusion . . . 15

3 Research Methodology 16
- 3.1 Introduction . . . 16
- 3.2 Research Hypothesis . . . 16
- 3.3 Motivation . . . 16
- 3.4 Method . . . 17
- 3.5 Conclusion . . . 17

4 Research Plan 18
- 4.1 Introduction . . . 18
- 4.2 Research Phases . . . 18
  - 4.2.1 Implementation . . . 18
  - 4.2.2 Experimentation . . . 19
  - 4.2.3 Analysis . . . 19
- 4.3 Time Plan . . . 19
- 4.4 Potential Issues . . . 20
- 4.5 Conclusion . . . 21

5 Conclusion **22**

List of Figures

2.1 Reinforcement learning agent-environment interaction cycle. 3
2.2 Platform domain 13
2.3 Robot Soccer Goal domain. 14

Chapter 1

Introduction

Reinforcement learning (RL) is a machine learning paradigm for control problems. It aims to mimic how humans appear to learn tasks by trial-and-error. Consider learning to score a goal in soccer. This problem requires knowing how to kick the ball to a certain position, inside the goals. A human would practise kicking the ball, experimenting with kicking at different angles and with different forces until some combination of angle and force results in the ball going into the goals. One would then try to perform subsequent kicks with a similar angle and force that previously resulted in a goal. Consider another soccer task: passing the ball to a teammate. While the underlying action of kicking the ball to a certain position is the same as scoring a goal, we know that the position kicked to needs to intercept the trajectory of the teammate if they are moving, and the force required, in general, should be less than that required when trying to score a goal. So while both tasks have the same underlying action, kicking the ball, how it is kicked changes depending on the task we want to achieve.

Previous work in reinforcement learning focused on problems with either discrete or continuous actions, which often don't reflect real-world problems, limiting the scope of RL. With discrete actions in RL, keeping with the soccer example, each kick with a different angle and force would be considered a separate action. With continuous actions, we try to optimise the angle and force components of the kick in all cases, but using this method a learning agent cannot easily differentiate a goal-kick from a pass, and so the learned parameters are unlikely to achieve either task. The kick action can instead be parameterised into two discrete actions: pass and goal-kick, each having the continuous angle and force parameters. Parameterised actions, discussed in Section 2.2.3, better represent how humans appear to learn tasks with parameters, allowing for domains that more accurately represent real-world problems to be solved. Such a parameterisation also lends itself well to robotics, where actuators require a mix of different high-level actions with low-level control in order to perform different tasks.

Reinforcement learning algorithms can generally be classified as either model-free or model-based. Model-free algorithms do not explicitly learn or represent the control dynamics of the environment. Going back to the soccer example, a model-free agent would not know exactly how the kick's angle and force actually changes the ball's position, only whether or not we expect the kick to result in a successful goal/pass. An agent using a model would know that kicking the ball with greater force will send it further away. So model-based algorithms try to make use of more of the available information and, in general, require less training to learn good action policies, see Section 2.3.

We will briefly review the foundational concepts and techniques of modern reinforcement learning in Chapter 2. We discuss several existing algorithms for discrete actions and continuous actions in Section 2.2, and

the recently introduced Q-PAMDP algorithm for parameterised actions in Section 2.2.3. We then examine how using model-based techniques that exploit knowledge of the environment, such as the Dyna framework, Section 2.3, can be used to improve the learning speed of certain algorithms.

In Chapter 3, we propose extending the Q-PAMDP algorithm with such a model-based framework in an effort to reduce the number of training episodes it requires to learn a good action policy. Lastly, in Chapter 4 we describe an overall plan for the implementation of the proposed research, along with a 4 month time plan.

Chapter 2

Background and Related Work

2.1 Introduction

In this chapter we discuss modern reinforcement learning techniques for Markov Decision Processes with different action spaces. For discrete actions, we focus primarily on the history of *temporal-difference* (TD) learning methods, which are characterised by using the difference between temporally successive estimates of the same quantity [Sutton and Barto 1998], and value-function based techniques. In Section 2.2.1, we discuss the use of function approximation to deal with continuous state spaces, which are the norm for most modern domains. Policy search methods for continuous action spaces are briefly discussed in Section 2.2.2, specifically policy gradient and actor-critic methods. Section 2.2.3 details the formulation of parameterised action spaces and the Q-PAMDP algorithm. The final part of the background, Section 2.3, discusses model-based reinforcement learning and the Dyna framework. The rest of the chapter then details existing parameterised action space domains and related research.

2.2 Reinforcement Learning

Reinforcement learning (RL) can be seen as a form of trial-and-error learning in which an agent learns to perform a task by maximising numeric rewards based on its actions performed in an environment. The agent repeatedly performs an action, transitions to a new state of the environment, and receives a reward, until a terminal state is reached that corresponds with the task being completed or failed. A *state* refers to the information given by the environment at a certain timestep. Considering the example of trying to score

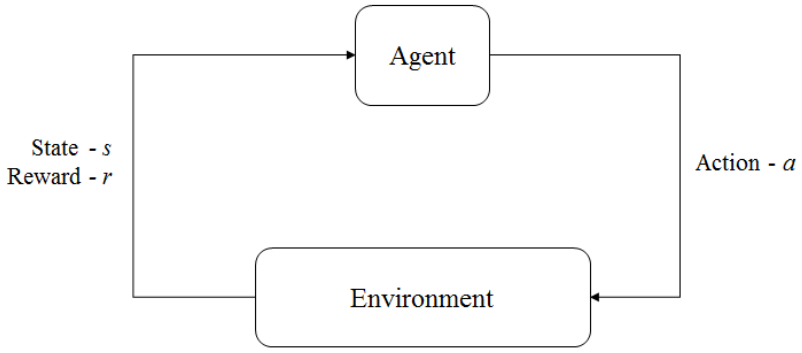

Figure 2.1: Reinforcement learning agent-environment interaction cycle.

a goal in soccer, the state could be the position of the ball, the position of the goals, and the distance of the

ball to the goals. The reward could be 1 for scoring a goal and -1 for missing.

Reinforcement learning algorithms typically consider domains where the state information sent to the agent contains all relevant information required to determine the next transition. This is the Markov property, which ensures the transition to a future state is dependent only on the action taken in the present state, and not on any preceding states. Such domains are formulated as *Markov Decision Processes* (MDPs). An MDP is a tuple, $M = (S, A, P, R)$ where S is the set of possible states, A is the set of actions available to the agent, $P(s'|s,a)$ is the probability of transitioning to state s' after taking action a in state s, and $R(s,a,s')$ is the expected scalar reward r obtained by taking action a in state s and transitioning to state s'. Note that the definition of the functions $P(s'|s,a)$ and $R(s,a,s')$ ensures that the Markov property is satisfied. [Sutton and Barto 1998]

The actions of an agent are determined by its *action policy* $\pi(a|s)$, which gives the probability of choosing action a in state s. Deterministic policies are denoted $\pi(s)$, the action chosen in state s. For now, we will assume that the set of actions A is finite and that each action is discrete. The aim of reinforcement learning algorithms is to find an action policy that maximises the cumulative discounted reward of the agent, given by

$$\sum_{t=o}^{\infty} \gamma^t R(s_t, a_t, s_{t+1})$$

where $0 \leq \gamma \leq 1$ is a discount factor chosen by the algorithm that determines how important future rewards are. State values, denoted by $V^{\pi}(s)$, represent the expected cumulative discounted reward under action policy π after reaching state s. The problem can then be seen as finding the solution to the *Bellman optimality equation*:

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')] \tag{2.1}$$

where $V^* = V^{\pi^*}$, and π^* is an optimal action policy. [Sutton and Barto 1998]

Value iteration is an algorithm that iteratively learns such an optimal value function. Starting from an arbitrary initialisation of V, the algorithm sweeps across all state values and updates them according to the Bellman optimality equation (2.1) at each iteration. A deterministic, optimal action policy is then given by greedily selecting the action that maximises V^*:

$$\pi(s) = \operatorname*{argmax}_a \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma V^*(s'))$$

Using this method, the greedy policy is guaranteed to converge in a finite number of steps even if the value function has yet to converge. [Bertsekas 1987]

Clearly, iterating over all possible states is slow, usually infeasible and assumes knowledge of the state transition and reward functions. State-action values, named *Q-values* and denoted by $Q^{\pi}(s,a)$, are similar to state values except they map state-action pairs. They represent the expected cumulative discounted reward for executing action a in state s and following the action policy π to the end of the episode. The Bellman optimality equation becomes:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')] \tag{2.2}$$

Using Q-values, one can learn a value function using samples gathered by the agent directly acting in the environment, without knowing the transition or reward functions of the domain.

Q-learning [Watkins and Dayan 1992] is a model-free TD algorithm that iteratively learns the optimal Q-value function for an MDP. The update rule is again derived from the Bellman equation (2.2): for each new sample $(s_t, a_t, r_{t+1}, s_{t+1})$ from the environment at time step t, if a was the greedy action then

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t[r_{t+1} + \gamma \max_{a'} Q_t(s_{t+1}, a') - Q_t(s_t, a_t)] \tag{2.3}$$

where $0 < \alpha_t \leq 1$ is the *learning rate*, or *step-size parameter*. An optimal action policy is then, for a state s, greedily selecting the action that maximises $Q^{\pi}(s,a)$. The update only occurs if the agent chooses the greedy action, making Q-learning an *off-policy* method. The learning rate may be fixed at some small value or reduce during learning; its purpose is to bound the magnitude of updates to prevent 'overshooting' the optimal value, as in gradient ascent/descent methods. This is particularly important to prevent divergence during value-function approximation updates, which are covered in Section 2.2.1.

2.2.1 Value Function Approximation

Value functions for finite discrete state spaces can usually be represented and learned using tabular methods. Continuous state spaces on the other hand, are composed of real-valued state variables, $S \subseteq \mathbb{R}^n$, making the number of states infinite. One approach is to discretise the continuous variables so that finite-state learning methods can be applied directly. Tile coding is a method that quantises each continuous variable into an exhaustive set of tiles covering the range of values. The features of a state are then binary values corresponding to the tiles into which each state variable falls. However, the resolution of the discretisation is an important consideration. Tilings that are too coarse risk losing information because it is not possible to distinguish between values grouped within a tile, which could make finding a good solution impossible. Tilings that are too fine-grained lack generalisation, since tabular methods require each state or state-action pair in a table to be visited in order to assign a value, requiring many more training samples to converge. Depending on the number of variables and resolution, the discretisation may also result in an intractably large state space that would make tabular value iteration methods infeasible. [Sutton and Barto 1998]

Parametric value-function approximation is another approach for cases where the value function cannot be represented exactly. We focus on linear value-function approximations, where a linear combination of features is used to represent the value function:

$$\hat{V}(s) = \sum_{i=1}^{k} w_i \phi_i(s) = \mathbf{w}^T \Phi(s)$$

where $\Phi = \{\phi_1, \phi_2, ..., \phi_k\}^T$ is a set of features, $\phi_i(s)$ is the value of feature i in state s, and $\mathbf{w} \in \mathbb{R}^n$ is the parameter or weight column vector [Parr *et al.* 2008]. The approximation for $\hat{Q}$ is similarly defined by making the features functions of state and action. Practically, this results in the features being indexed by action. Parametric function approximation generalises to unexplored states since similar states produce similar values from the function, so fewer training samples are required. Hence function approximation avoids the time, memory, and data requirements of large value tables. [Sutton and Barto 1998]

A major problem in function approximation is how to choose the feature set. The representational power of the features chosen determines the quality of the approximation, and hence the performance of the derived policy. The terms of the feature set can be the state variables themselves, however most value functions are too complex to be represented as combinations of the state variables alone, so functions of the variables are used to form some basis. Radial basis functions (RBFs) are one option, whereby each feature is a Gaussian function centred at some fixed state value and width. They can be considered the natural generalisation

of coarse coding to continuous-valued features but produce real values in the interval [0,1] indicating the degree to which the feature is present instead of binary values [Sutton and Barto 1998]. The Fourier basis proposed by Konidaris *et al.* [2011] is another option, where terms of the Fourier series over different combinations of the state variables are used up to some order.

The next problem is finding the parameter vector $\mathbf{w}$ such that $\hat{V} \approx V^*$. The error of an approximate value function is usually analysed in terms of the one-step lookahead error, otherwise known as *Bellman error* [Parr *et al.* 2008]:

$$BE(\hat{V}) = r + \gamma P\hat{V} - \hat{V} \tag{2.4}$$

Note that the TD error used in the Q-learning update rule, Equation (2.3), is very similar. In general, a perfect set of weights cannot be found to exactly approximate the function over all states since the basis has limited resolution. Gradient descent methods are typically employed for iterative learning of $\mathbf{w}$ by minimising the Bellman error. Assuming a fixed number of features and that the approximate value function $\hat{V}$ is a smooth differentiable function of $\mathbf{w}$ for all $s \in S$, the update rule for TD(λ), a gradient descent algorithm, at time step t and sample $(s_t, a_t, r_{t+1}, s_{t+1})$ is:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \delta_t \mathbf{e}_t \tag{2.5}$$

where δ is the TD error, based on the Bellman error from Equation (2.4):

$$\delta_t = r_{t+1} + \gamma\hat{V}(s_{t+1}) - \hat{V}(s_t)$$

and $\mathbf{e} \in \mathbb{R}^k$ is the vector of eligibility traces for each component of $\mathbf{w}_t$. Eligibility traces are a technique of increasing the learning rate by temporal credit assignment, updating the previous states encountered along the trajectory of samples to the current state. The update rule for $\mathbf{e}$ is:

$$\mathbf{e}_{t+1} = \lambda \mathbf{e}_t + \nabla_{\mathbf{w}_t}\hat{V}(s_t)$$

where $\mathbf{e}_0 = \mathbf{0}$ and $0 \leq \lambda \leq 1$ is the exponential eligibility trace decay. When $\lambda = 0$, the update rule for TD(0) reduces to a more familiar form of gradient descent:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \delta \nabla_{\mathbf{w}_t}\hat{V}(s_t)$$

A useful property of linear function approximators is that the derivative of $\hat{V}(s_t)$ with respect to the weight vector $\mathbf{w}$ is just the feature vector at that state:

$$\nabla_{\mathbf{w}_t}\hat{V}(s) = \Phi(s)$$

This property greatly simplifies the update computation and is one reason for the popularity of linear function approximations. [Sutton and Barto 1998]

Sarsa(λ)

Sarsa(λ) is an iterative learning algorithm that applies TD(λ) to Q-values. Samples used in updates now include the action chosen for the next state, $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. The update step is the same as in Equation (2.5) but delta changes to:

$$\delta_t = r_{t+1} + \gamma\hat{Q}_t(s_{t+1}, a_{t+1}) - \hat{Q}_t(s_t, a_t)$$

and

$$\mathbf{e}_{t+1} = \gamma\lambda \mathbf{e}_t + \nabla_{\mathbf{w}_t}\hat{Q}(s_t, a_t)$$

The main distinction between this and Watkin's Q-learning, which can similarly be extended with eligibility traces and TD(λ) updates for value function approximation, is that Sarsa(λ) is *on-policy*: its updates occur whether or not the action chosen was greedy. This affects the learned policy because it takes into account any randomness of the agent's action selection, such as in ε-greedy policies. [Sutton and Barto 1998]

TD(λ) algorithms have two disadvantages, they make inefficient use of data, and require manual tuning of the step-size parameter α to prevent function approximation divergence. Least-Squares TD(λ), or LSTD(λ), introduced by Bradtke and Barto [1996] eliminates step-size parameters and converges to the same weight vector as TD(λ). TD(λ) can be viewed as using stochastic gradient descent to solve a system of equations $\mathbf{d} + \mathbf{C}\mathbf{w} = \mathbf{0}$ over T training samples, with

$$\mathbf{d} = \mathbb{E}\left\{ \sum_{t=0}^{\mathrm{T}} \mathbf{e}_t r_t \right\}$$

and

$$\mathbf{C} = \mathbb{E}\left\{ \sum_{t=0}^{\mathrm{T}} e_t [\Phi(s_{t+1}) - \Phi(s_t)]^T \right\}$$

where $\mathbf{e}$ is the eligibility trace vector. Algorithms such as TD(λ) and LSTD(λ) that solve the system $\mathbf{d} + \mathbf{C}\mathbf{w} = \mathbf{0}$ are collectively referred to as *fixed-point methods*. TD(λ) never represents $\mathbf{d}$ or $\mathbf{C}$ directly, using only the most recent sample to update $\mathbf{w}_t$. LSTD(λ) builds explicit representations of $\mathbf{d}$ and $\mathbf{C}$ over n iterations as a vector $\mathbf{b}$ and a $k \times k$ matrix $\mathbf{A}$ respectively, where k is the number of features:

$$\mathbf{b} = \sum_{t=0}^{n} \mathbf{e}_t r_t$$

$$\mathbf{A} = \sum_{t=0}^{n} \mathbf{e}_t [\Phi(s_{t+1}) - \Phi(s_t)]^T$$

The solution for $\mathbf{w}$ is then calculated directly by:

$$\mathbf{w} = \mathbf{A}^{-1}\mathbf{b} \tag{2.6}$$

This is known as the *linear fixed-point solution*. While least-squares algorithms require more computation, they use more of the available information from samples and can be expected to converge with fewer training samples than TD(λ), making them more data efficient. LSTD(λ) can also be interpreted as a model-based method because it effectively records a model of all the observed transitions and uses it to solve for the parameter vector. [Boyan 2002]

2.2.2 Continuous Action Spaces

Value-function based reinforcement learning algorithms usually index value functions by the discrete actions, making choosing an action a process of scanning through each action to determine which one maximises the current value or state-action value function. However, this is no longer possible with continuous action spaces. Continuous actions have values that lie in some real-valued and possibly multidimensional range, $a \in \mathbb{R}^m$. Continuous action spaces can, as with continuous state spaces and with similar disadvantages, be discretised such that RL algorithms for discrete actions can be applied directly. A fine-grained discretisation is usually required to obtain a good solution because the optimal action value may be 'hidden' between discrete values. Though creating too many discrete actions can significantly slow learning since

more actions have to be explored. Note that with multidimensional continuous actions, the number of evenly spaced discrete actions needed to cover all combinations grows exponentially with respect to the number of dimensions. Although we do not detail them in this document, there exist algorithms that attempt to alleviate these concerns using variable resolution discretisations of the continuous action space that don't need to be stored explicitly, such as Binary Action Search [Pazis and Lagoudakis 2009]. Those algorithms, however, still depend on the chosen resolution. An alternative is to learn a continuous action policy directly.

Policy Search

Policy search represents a class of algorithms separate to value-function based reinforcement learning. Instead of learning a value function that determines state quality and using it to derive a policy, an action policy is learned directly without discretising the action value range. Like value function approximations, however, the policy can be represented as as a linear combination of basis functions:

$$\pi(s) = \theta^T \Phi(s) \tag{2.7}$$

We use θ to represent the parameter vector for the action policy to distinguish it from the value-function parameter vector $\mathbf{w}$, although it serves a similar purpose. We denote an action policy π using such a representation under weight vector θ by π_θ. Note that the action policy can be stochastic as noise, commonly Gaussian in the continuous case, can be added to the deterministic result of $\pi(s)$.
A trajectory, denoted τ, is a collection of samples (s,a,r,s') gathered over an episode. The accumulated reward, or *return*, for a trajectory is given by

$$R(\tau) = \sum_{t=0}^{\mathrm{T}} r_t(s_t, a_t)$$

Policy search methods attempt to update the policy's parameter vector based on these sampled trajectories such that trajectories with higher accumulated rewards become more likely. This increases the excepted return given by:

$$J_\theta = \mathbb{E}[R(\tau)|\theta] = \int_\tau P_\pi(\tau) R(\tau) d\tau \tag{2.8}$$

where $P_\pi(\tau)$ is the distribution over the trajectories based on the current action policy:

$$P_\pi(\tau) = P(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, \pi_\theta(a_t|s_t))$$

[Deisenroth *et al.* 2013]

Policy Gradient

Policy gradient is a class of methods for finding θ. The parameter vector is learned iteratively using gradient ascent to maximise the expected return function J_θ. The update rule for θ at time t is then:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J_\theta \tag{2.9}$$

where $\nabla_\theta J_\theta$ is the policy gradient defined by:

$$\nabla_\theta J_\theta = \int_\tau \nabla_\theta P_\theta(\tau) R(\tau) d\tau$$

The next problem is how to calculate $\nabla_\theta J_\theta$. There are a variety of ways to estimate the policy gradient. One such method is the *Episodic Natural Actor Critic* (eNAC) algorithm. It calculates the natural gradient, which optimises parameterised probability distributions faster than using the traditional gradient. The natural gradient uses a Fisher information matrix to account for successive policies possibly causing large differences to the trajectory distribution:

$$F_\theta = \mathop{\mathbb{E}}_{P(\tau)}\left[\nabla_\theta log P_\theta(\tau) \nabla_\theta log P_\theta(\tau)^T\right]$$

The eNAC policy gradient is then:

$$\begin{aligned}\nabla_\theta^{eNAC} J_\theta &= F_\theta^{-1} \nabla_\theta J_\theta \\ &= \mathop{\mathbb{E}}_{P(\tau)}\left[\Big(\sum_{t=0}^{T-1} \nabla_\theta log \pi_\theta(a_t|s_t)\Big)\Big(\sum_{t=0}^{T-1} \nabla_\theta log \pi_\theta(a_t|s_t)\Big)^T\right]^{-1} \mathop{\mathbb{E}}_{P(\tau)}\left[\sum_{t=0}^{T-1} \nabla_\theta log \pi_\theta(a_t|s_t) R(\tau)\right]\end{aligned}$$

The eNAC algorithm estimates the expectations of these terms by averaging the summations over multiple episodes using the current action policy π_θ. [Deisenroth *et al.* 2013]

Note that eNAC requires full episodic roll-outs to perform each update, as opposed to TD methods which update at each time step of an episode. This makes policy gradient algorithms ill-suited to domains with very long episodes or unending tasks.

Incremental Actor-Critic

Actor-critic reinforcement learning algorithms are a combination of value function and policy based methods. Such algorithms are comprised of two parts: a critic that learns a value function, and an actor that uses feedback from the critic in order to update an action policy. The advantage of such an approach over direct policy gradient is that the actor-critic framework can sample more efficiently using temporal difference updates at each step, rather than requiring several trajectories to estimate the gradient. [Deisenroth *et al.* 2013]

For incremental actor-critic algorithms, we consider maximising the expected discounted return over an episode:

$$J_\theta^\gamma = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t(s_t, a_t) | \pi_\theta] \tag{2.10}$$

Consider the state-action value function $Q^\pi(s,a)$, which as discussed in Section 2.2 gives the expected discounted return under policy π. Using a linear approximator for the action policy, Equation (2.7), we can construct a compatible linear function approximation of Q^π:

$$\hat{Q}^{\pi_\theta}(s,a) = \mathbf{w}^T \psi^{\pi_\theta}(s,a) \tag{2.11}$$

where $\psi^{\pi_\theta} = \nabla_\theta log \pi_\theta(s,a)$ are *compatible features* based on the features used in the action-policy representation, π_θ. Then the gradient of the discounted expected return from Equation (2.10) is simply:

$$\nabla_\theta J_\theta^\gamma = \tilde{\mathbf{w}} \tag{2.12}$$

where $\tilde{\mathbf{w}}$ minimises the mean-squared error of the approximation

$$\varepsilon^{\pi_\theta}(\mathbf{w}) = \sum_{s,a} d^{\pi_\theta}(s) \pi_\theta(a|s) \left[\mathbf{w}^T \psi^{\pi_\theta}(s,a) - Q^{\pi_\theta}(s,a)\right]^2 \tag{2.13}$$

and $d^{\pi_\theta}(s)$ is the stationary distribution of state s under policy π. [Kakade 2001]

Using the approximate action-value function from Equation (2.11) as our critic, and the gradient of the discounted expected return, Equation (2.12), the action policy update can be performed using the policy gradient update rule, Equation (2.9). Next, we consider the problem of finding such an optimal $\tilde{\mathbf{w}}$.

Natural Actor Critic using Sarsa(λ)

An objective function can be defined based on the discounted expected return and mean squared error:

$$\mathbf{H}(\mathbf{w}) = \sum_{t=0}^{\infty} \mathbb{E}_{s,a}\left[\frac{\gamma^t}{2}\left(\mathbf{w}^T \psi^{\pi_\theta}(s,a) - Q^{\pi_\theta}(s,a)\right)^2\right] \tag{2.14}$$

where the state-action value is estimated from the Monte Carlo return $Q^{\pi_\theta}(s_t,a_t) = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ under π_θ. If $\tilde{\mathbf{w}}$ minimises Equation (2.14), then so too does it minimise the mean-squared error from Equation (2.13). To overcome having to sample entire trajectories, temporal difference updates are used with Sarsa(λ) eligibility traces to optimise $\mathbf{w}$ subject to Equation (2.14). This leads to the following update rule at time t for sample $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$:

$$\begin{aligned}
\mathbf{e}_{t+1} &= \gamma\lambda e_t + \gamma^t \nabla_{\mathbf{w}} \hat{Q}^{\pi_\theta}(s_t,a_t) \\
\delta_t &= r_{t+1} + \gamma \hat{Q}^{\pi_\theta}(s_{t+1},a_{t+1}) - \hat{Q}^{\pi_\theta}(s_t,a_t) \\
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha_t^{\mathbf{w}} \delta_t \mathbf{e}_t \\
\theta_{t+1} &= \theta_t + \alpha_t^{\theta} \frac{\mathbf{w}_{t+1}}{||\mathbf{w}_{t+1}||_2}
\end{aligned}$$

Using this update rule gives us the *Natural Actor Critic using Sarsa(λ)*, or NAC-Sarsa(λ), algorithm, which iteratively updates the critic's state-action value function approximation using Sarsa(λ), see Section 2.2.1, and updates the action policy using stochastic gradient ascent, as with policy gradient. The algorithm is proven to converge to at least a locally optimal policy with respect to J, since it ascends the natural gradient of J. [Thomas 2014]

2.2.3 Parameterised Action Spaces

Parameterised action spaces consist of a set of discrete actions, $A_d = \{a_1, a_2, ..., a_n\}$, where each a_i has corresponding continuous parameters $x_i \in R_i^m$. Markov decision processes with such action spaces are referred to as parameterised action Markov decision processes (PAMDPs). For a PAMDP $M = (S, A, P, R)$, the action space A is composed of actions with their parameter spaces, and the transition probability function $P(s'|s,a,x)$ and reward function $R(s,a,x,s')$ depend on the action and its parameter. [Masson *et al.* 2016]

Once again we consider the naïve approach of discretisation. Each continuous parameter can be discretised to some resolution in order to create a finite number of action-parameter pairs. Clearly, this technique can quickly create an intractably large action space that would make discrete reinforcement learning methods infeasible. Thus discretisation is inappropriate for parameterised action spaces.

Another approach is to apply a direct policy search, such as the eNAC policy gradient algorithm, to optimise the action policy with respect to the expected return. While this method is technically feasible, directly optimising over the entire action space can take a long time, and may converge early to a bad local optimum. [Deisenroth *et al.* 2013]

Q-PAMDP

Q-PAMDP(k), Algorithm 1, proposed by Masson *et al.* [2016] is a model-free reinforcement learning method for parameterised action spaces. Instead of learning the policy $\pi(a,x|s)$ to select both a discrete action and its parameter at once, the policy is split into a policy for discrete action selection, denoted $\pi^d(a|s)$, and another policy for action-parameter selection, $\pi^a(x|s)$, which is indexed by the action a. A parametric value-function approximation with weight vector $\mathbf{w}$ is used for the discrete action policy, so the action policy using $\mathbf{w}$ is denoted $\pi^d_{\mathbf{w}}(a|s)$. Similarly, a parametric function approximation with weight vector θ is used for the action-parameter policy, denoted $\pi^a_\theta(x|s)$. Then for a PAMDP, $M = (S,A,P,R)$, the discrete MDP using a fixed action-parameter policy with a certain weight vector θ is given by $M_\theta = (S,A_d,P_\theta,R_\theta)$, where A_d is the discrete action set. The transition and reward functions $P_\theta(s'|s,a)$ and $R_\theta(s,a,s')$ act as $P(s'|s,a,x)$ and $R(s,a,x,s')$, but with x determined by the fixed action-parameter policy π^a_θ.

Q-PAMDP comprises a Q-learning algorithm for learning $\mathbf{w}$, and an algorithm such as policy search for learning the parameter policy weight vector θ by optimising the objective function

$$J(\theta,\mathbf{w}) = \mathop{\mathbb{E}}_{s_0}[V^\pi(s_0)] \tag{2.15}$$

where s_0 is a state sampled according to the initial state distribution of the MDP, and V is a value function. With $\mathbf{w}$ fixed, Equation (2.15) is denoted $J_{\mathbf{w}}(\theta)$. The algorithms are alternated each iteration, with the action policy fixed while the parameter is updated for k steps before fixing the parameter policy and updating the action policy to convergence. In Algorithm 1, P-UPDATE is the algorithm chosen for updating the parameter policy weight vector θ with respect to $J_{\mathbf{w}}(\theta)$, and Q-LEARN is any Q-learning function with parametric function approximation, it does not refer specifically to Watkin's Q-learning algorithm.

Algorithm 1 Q-PAMDP(k)

Input: Q-LEARN, P-UPDATE, and initial vectors θ_0, $\mathbf{w}_0$
  $\mathbf{w} \leftarrow$Q-LEARN$^{(\infty)}(M_\theta, w_0)$
  repeat
    $\theta \leftarrow$P-UPDATE$^k(J_{\mathbf{w}}(\theta),\theta)$
    $\mathbf{w} \leftarrow$Q-LEARN$^{(\infty)}(M_\theta, w_0)$
  until θ converges

Masson *et al.* [2016] proved the algorithm converges for the two extreme cases of $k = 1$ and $k = \infty$. QPAMDP(1) converges to a locally optimal solution with respect to the expected reward, provided the choice of P-UPDATE algorithm converges to a local or global optimum. However, QPAMDP(∞), which updates θ to convergence each iteration, requires that P-UPDATE converges to a global optimum for the algorithm to converge locally. Using Sarsa(λ) for Q-LEARN and eNAC for P-UPDATE, both cases were shown to outperform both a discrete Sarsa(λ) agent using a fixed action-parameter policy, and a direct policy search agent using only eNAC. This result indicates that a full search over the action-parameter space is not necessary to learn a good policy.

We note that the choice of the policy gradient algorithm, eNAC, for P-UPDATE makes the overall algorithm similarly ill-suited to domains with very long episodes or unending tasks (see Section 2.2.2).

2.3 Model-Based Reinforcement Learning

All the techniques discussed hitherto have been model-free, that is, they do not explicitly learn or store the environment's dynamics in the process of learning an action policy. Model-based reinforcement learning algorithms construct a representation of the environment's control dynamics. This requires determining or approximating the transition function P and reward function R of the MDP. We will review techniques that use the model to simulate transitions for use in value function updates. This can accelerate learning compared to model-free methods since multiple transitions can be simulated per episode, including unexplored transitions. This makes model-based techniques more data efficient, so they generally converge to a good policy with fewer training samples than required by model-free techniques. [Kuvayev and Sutton 1996]

2.3.1 Dyna

The *Dyna* architecture proposed by Sutton [1990] uses a model to substitute transition samples from the environment for model-free reinforcement learning algorithms. During learning, a *planning* step takes a state that was previously explored, or generated from some distribution, and predicts the next state and reward, this information is then used in the usual update step of the model-free algorithm. For discrete, table-based state spaces, the model can be represented as a hashmap for each state-action pair (s,a), which contains the frequencies of past occurrences of transitions to each s' after taking action a in state s. The transition probability can then be estimated directly based on these frequencies. Employing tile coding can extend this method to continuous state spaces by grouping similar s' values. However this still suffers from the same disadvantages discussed in Section 2.2.1; we cannot generalise to unexplored transitions using this method. [Kuvayev and Sutton 1996]

Sutton *et al.* [2008] introduced the *Linear Dyna* algorithm to extend Dyna from discrete, tabular state spaces to continuous state spaces using linear function approximation. The algorithm is based on *Dyna-Q*, which is simply Q-learning enhanced with the Dyna architecture. A model is learned and used for planning updates at the same time that updates from real experience samples are performed. The model in Linear Dyna is approximated using the same feature set Φ used for the approximate value-function. A forward transition matrix $F \in \mathbb{R}^k \times \mathbb{R}^k$ and an expected reward vector $\mathbf{b} \in \mathbb{R}^k$ are formed such that $\Phi(s') = F\Phi(s)$ and $r = \mathbf{b}^T\Phi(s)$, where Φ is a column vector and k is the number of features. The model generates the expectation of the next feature vector and reward, and is therefore deterministic even if it approximates a stochastic environment. Stochastic gradient descent is used to update F and $\mathbf{b}$. At time step t for a real sample $(s_t, a_t, r_{t+1}, s_{t+1})$:

$$F_{t+1} = F_t + \alpha(\Phi(s_{t+1}) - F\Phi(s_t))\Phi(s_t)^T$$

$$\mathbf{b}_{t+1} = \mathbf{b}_t + \alpha(r_{t+1} - \mathbf{b}_t^T\Phi(s_t))\Phi(s_t)$$

Note that the transition matrix predicts the features of the next state, rather than the state variables. Linear value-functions are approximated by $\mathbf{w}^T\Phi(s)$ so the predicted features can be used directly during the value function update.

Using the model defined above, the parameter vector $\mathbf{w}$ for the value-function approximation can be calculated directly as follows:

$$\mathbf{w} = (I - \gamma F)^{-1}\mathbf{b} \tag{2.16}$$

The solution to Equation (2.16) is referred to as the *linear model solution*. Parr *et al.* [2008] and Sutton *et al.* [2008] prove that the linear model solution is equivalent to the fixed-point solution, Equation (2.6), found

by LSTD. Parr *et al.* [2008] also prove that $\mathbf{w}$ learned using Linear Dyna converges to this solution.

2.4 Domains

Masson *et al.* [2016] introduced two domains with parameterised actions and continuous state spaces: the Platform domain, and a 2D Robot Soccer Goal domain. We discuss in Chapter 3 how we intend to evaluate our proposed model-based Q-PAMDP algorithm on these domains against the original Q-PAMDP algorithm.

2.4.1 Platform Domain

The Platform domain consists of a 2-dimensional game-world with stationary platforms, moving enemies, and the agent. In the default layout, each platform has a single enemy which patrols back and forth at a constant speed. The enemies cannot jump and move only when the agent is within the horizontal range of the platform; this is a simplification that restricts the feature set to consider only a single enemy's horizontal position and velocity, $(x_e, \dot{x_e})$, at a time. Two basic actions are available to the agent: run and jump, which continue for a fixed period or until the agent lands again respectively. The jump is parameterised into two separate actions: a hop to get over enemies, and a leap to traverse the gaps between platforms. Initially, the leap parameter is too small to clear the gaps between the platforms. The reward function is based on the normalised distance the agent moves towards the goal, so the rewards obtained over a successful episode sum up to 1. An episode ends when the agent touches an enemy, falls into a gap between two platforms, or reaches the goal.

Figure 2.2: Starting configuration of the Platform domain with three platforms. The agent always starts on the left of the first platform. Each enemy starts on the right edge of its platform. The goal is to reach the end of the third platform on the right.

The basic feature set for learning the discrete action selection is composed of the agent's horizontal position and velocity and that of the enemy on the current platform, $(x, \dot{x}, x_e, \dot{x_e})$. An enhanced feature set is available for learning the action-parameters which includes the horizontal position of the current platform, x_p, the width of the current and succeeding platform, w_p and $w_{p'}$, and the gap and height between the current and next platform, g and h respectively. This gives $(x, \dot{x}, x_e, \dot{x_e}, x_p, w_p, w_{p'}, g, h)$ as the enhanced feature set. In the default configuration, however, h is always 0 because all platforms are at the same height. [Masson *et al.* 2016]

2.4.2 Robot Soccer Goal Domain

The Robot Soccer Goal domain is based on the 2D robot soccer challenge domain. This simplified domain focuses on a particular task in soccer: scoring by shooting the ball past the goal keeper and into the goals. Two basic actions are available to the agent: kick-to(x, y), which kicks to ball towards position (x, y) on the field; and shoot-goal(h), which shoots the ball towards a position h along the goal line. Gaussian noise

is added to each action. The shoot-goal action is split into two parameterised actions: shoot-goal-left and shoot-goal-right. This is because the agent has to shoot around the keeper to score a goal, either to the left or right of it but never directly at it. The agent automatically moves towards the ball whenever it is not in possession of it. The keeper either moves towards the ball or, if the agent shoots at the goal, moves to intercept the ball.

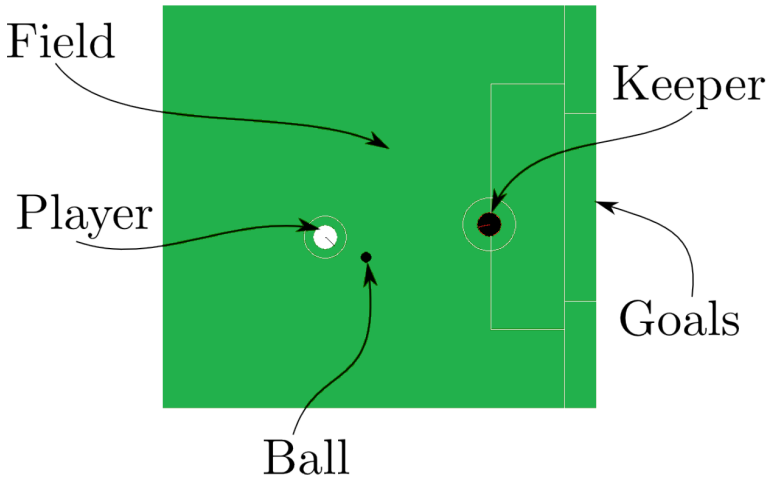

Figure 2.3: Robot Soccer Goal domain.

The agent starts in possession of the ball, at a random position along the left bound of the field. The keeper is positioned between the ball and the goal. The agent has a position (x, y), velocity $(\dot{x}, \dot{y})$ and orientation (θ), as does the keeper. The ball also has a position and velocity, so each state has 14 variables: $(x, y, \dot{x}, \dot{y}, \theta, x_k, y_k, \dot{x}_k, \dot{y}_k, \theta_k, x_b, y_b, \dot{x}_b, \dot{y}_b)$, where the subscripts k and b distinguish the variables of the keeper and ball respectively. An episode ends when the keeper intercepts the ball, the agent scores a goal, or the ball leaves the field. The agent receives a reward of 50 scoring a goal, $-d$ when the ball is intercepted or leaves the field, where d is the distance of the ball to the goal, and 0 otherwise. [Masson *et al.* 2016]

2.5 Related Work

Hausknecht and Stone [2016] apply the formulation of parameterised action Markov decision processes to extend the Deep Deterministic Policy Gradients (DDPG) algorithm. The authors use an actor-critic artificial neural network architecture with an inverting gradients technique to bound the gradient-based updates for stability. To deal with parameterised actions, two layers of artificial neurons are used to represent and learn the discrete actions and each of the associated action-parameters separately, before being combined into a final inner product layer. The authors evaluated their algorithm on a 2D simulated RoboCup soccer domain, of which the Robot Soccer Goal domain presented by Masson *et al.* [2016] is a simplified version.

Khamassi and Tzafestas [2016] extend the actor-critic neural network learning algorithm for PAMDPs presented by Hausknecht and Stone [2016] with active exploration in order to increase the algorithm's learning speed. Active exploration introduces hyperparameters that balance exploration versus exploitation throughout learning. The authors show that this technique can improve the speed of convergence in the case of parameterised action spaces.

We note that both of the above papers use model-free artificial neural network algorithms. To the best of our knowledge, no research has been done in model-based learning for domains with parameterised action spaces.

2.6 Conclusion

Q-PAMDP is the first provably convergent algorithm to extend previous model-free reinforcement learning techniques to parameterised action spaces, Section 2.2.3. Masson *et al.* [2016] experimented with using a policy gradient parameter update algorithm as part of Q-PAMDP, which has the disadvantage of requiring episodic roll-outs to calculate gradients, making it data inefficient and ill-suited to domains with very long episodes or unending tasks. We noted in Section 2.3 that model-based algorithms are generally more data efficient, requiring fewer training episodes to converge. In Chapter 3, we propose accelerating the Q-PAMDP algorithm's learning speed by using an incremental actor-critic algorithm integrated with a model-based framework.

Chapter 3

Research Methodology

3.1 Introduction

The objective of this research is to improve the learning speed of the Q-PAMDP algorithm using model-based techniques. Specifically, we propose substituting the data-inefficient policy gradient algorithm used in its parameter update step by an incremental actor-critic method, and integrating the model-based Dyna framework. We intend to use this proposed algorithm to investigate the following hypothesis:

3.2 Research Hypothesis

- Using a model-based reinforcement learning algorithm for the parameter update step of the Q-PAMDP algorithm will reduce the number of training episodes required for the algorithm to converge to a good action policy.

3.3 Motivation

As noted in Section 2.2.2, the model-free policy gradient algorithm eNAC, used by Masson *et al.* [2016] for the continuous parameter update step of Q-PAMDP, has the disadvantage of requiring episodic roll-outs to calculate gradients. This makes the overall algorithm data inefficient and only suitable for domains with short, episodic tasks. Incremental actor-critic algorithms, such as the model-free NAC-Sarsa(λ), do not require episodic roll-outs and can update at each step of an episode. Hence it makes sense that substituting eNAC with NAC-Sarsa(λ) could increase the learning speed of Q-PAMDP and make it suitable for a larger variety of domains. Furthermore, model-based reinforcement learning techniques are generally more data efficient, and so it stands to reason that using a model-based framework such as Dyna for Q-PAMDP would further increase its learning speed.

There are two obvious concerns with the proposed research: whether NAC-Sarsa(λ) will work for the parameter update step of Q-PAMDP, and whether the Dyna framework can be used with NAC-Sarsa(λ). From Section 2.2.3, the convergence properties of Q-PAMDP hold in general for parameter update algorithms that optimise the expected return. Incremental actor-critic algorithms are proven to converge to at least a locally optimal solution, Section 2.2.2, and hence should be suitable for the parameter update step of Q-PAMDP. Next, we note that NAC-Sarsa(λ) uses a linear approximation of the state-action value function for its critic. Linear Dyna specifically uses a linear model representation of the forward transition function, using the

same features as the linear value function approximator. NAC-Sarsa(λ) uses temporal difference updates based on Sarsa(λ), and hence TD(λ). This means it obtains a linear fixed-point solution for the weight vector of the linear function approximation of the state-action value function. Since the linear model solution found by Dyna for the weight vector is proven to be equivalent to the linear fixed-point solution [Parr *et al.* 2008], it follows that Dyna-style planning updates can be applied.

3.4 Method

The hypothesis will be tested experimentally. We will compare the proposed model-based algorithm with two model-free Q-PAMDP versions, one that uses eNAC for its parameter update step, and one that uses NAC-Sarsa(λ) for its parameter update step. We include the model-free Q-PAMDP algorithm using NAC-Sarsa(λ) in the comparison because changing the parameter update algorithm from eNAC may cause a change in the convergence speed which cannot be attributed to using a model-based technique. The three algorithms will be evaluated on the original parameterised action space domains presented by Masson *et al.* [2016], the Platform and Robot Soccer Goal domains described in Section 2.4. On each domain, we will compare the algorithms in terms of the number of episodes taken to converge and the average episodic return obtained. We include the second metric in order to determine whether the model-based algorithm affects the quality of the resulting action policy, because converging sooner to a poor action policy would not be useful.

3.5 Conclusion

The objective of this research, using model-based methods to improve the convergence speed of the Q-PAMDP algorithm, was established and motivated in this chapter. We proposed two alterations to the algorithm, namely using an incremental actor-critic method for its parameter update step, and applying the model-based Dyna framework to said method to increase its learning speed. We also outlined the experimental methodology by which we intend to verify whether or not our proposed algorithm has a faster convergence speed relative to model-free Q-PAMDP. We discuss the intended execution of this research in the next chapter.

Chapter 4

Research Plan

4.1 Introduction

Chapter 3 introduced the research hypothesis and method by which we intend to test it. We now consider the practical execution of this research and present a time plan governing our intended progress. Lastly, we consider possible issues which may arise during the course of this project.

4.2 Research Phases

We detail the three main phases of this research, namely implementation, experimentation, and analysis.

4.2.1 Implementation

The first step is implementing the code for the reinforcement learning algorithms and the domains used in the experiment. Fortunately, Masson *et al.* [2016] released Python code for their domains and the Q-PAMDP algorithm using Sarsa(λ) and eNAC, available on GitHub*. So this initial process simplifies to extending the parameter update step of Q-PAMDP using the NAC-Sarsa(λ) algorithm and Dyna framework.

In order to ensure the results obtained from experimentation are valid, each individual part of the proposed algorithm needs to be tested. The NAC-Sarsa(λ) algorithm will be tested for correctness on a domain with a continuous action space, such as Cart Pole†. Next, NAC-Sarsa(λ) will be extended with the Dyna framework. In order to proceed with reasonable confidence to testing on parameterised action spaces, the NAC-Sarsa(λ) with Dyna algorithm ideally should show some improvement over the model-free version in terms of convergence speed. Once this is established, we can make implement our two variants of Q-PAMDP by substituting the parameter update step with NAC-Sarsa(λ) and NAC-Sarsa(λ) with Dyna.

The final step of the implementation stage is a hyperparameter search. The actor-critic algorithm has two distinct learning rates, $\alpha^{\mathbf{w}}$ and α^{θ}, for the critic and actor respectively and a discount factor, γ, each of which can significantly impact its performance. Notably, a learning rate that is too high can cause the function approximation to diverge, and too low will not learn fast enough over a complete training session. With the integration of an online-learned model for Dyna, we introduce a third learning rate for the model updates. We thus expect the model-based algorithm to be even more sensitive to the hyperparameter values. A search will be performed for the two variants of Q-PAMDP using NAC-Sarsa(λ). That is, we use several different

*https://github.com/WarwickMasson/aaai-platformer and https://github.com/WarwickMasson/aaai-goal

†Classic continuous action space benchmark domain that involves trying to balance an upright pole on a moving cart.

combinations of hyperparameter values, each within some reasonable range, and test the algorithms on the experiment domains. The values for which the algorithms perform best will then be chosen for use in the comparison experiments.

4.2.2 Experimentation

The second stage is performing the actual experiments to compare the performance of the algorithms. Given that the code for the domains is already available, only minor adjustments should be required to set up the code to perform the experiments. There are two steps to the experiments: training, and policy evaluation. As mentioned in Chapter 3, we intend to evaluate the convergence speed of each algorithm and the converged policy. To do this, we need to average the number of episodes taken for the policy to converge in each case over several training sessions on both domains. Then, we evaluate the average episodic return of the converged policies, which requires fixing the state of the learning algorithm after training and recording the total reward the policy obtains, averaged over several episodes.

4.2.3 Analysis

The third and final stage of this research is analysing the results and writing the final report. The analysis includes interpreting the obtained results and may require going back and retuning the hyperparameters for further experimentation if unexpected behaviour of the algorithms is observed.

4.3 Time Plan

We present a weekly time plan in Table 4.1, with week dates specified starting from Monday. The allocated time for each task is specified per week. We formally start our research plan from 20 June, by which mid-year exams are finished. We allocate a significant portion of time to the initial implementation and testing of the NAC-Sarsa(λ) and Dyna extended algorithms so that any potential problems with the choice of algorithm, see Section 4.4, can be discovered and addressed in good time. The hyperparameter search is allocated two weeks, since there are several different variables to consider in addition to the learning rates, such as the number of parameter updates to perform per iteration of the Q-PAMDP algorithm, each of which requires testing in combination with every other hyperparameter value in the search. The time will primarily be spent queuing cluster jobs with different hyperparameter values and analysing the results.

During our preliminary analysis of the original Q-PAMDP code, we found that an entire training session can take around one hour. Given that learning a model online is more computationally expensive, we budget around 2 hours per training session. Assuming we take our averages over 20 complete training sessions per algorithm per domain, we estimate a total of 240 computational hours for the final experiments. With the use of the University of the Witwatersrand's Mathematical Sciences cluster, we are confident that the final experiments can be finished within one week, but allocate two weeks to be on the safe side. By that point, the execution of experiments on the cluster and results extraction should be automated, we dedicate two weeks in August to setting up this final process and ensuring everything works.

Week Starting	Task	Allocated Time (Hours)
June 19	Implementing NAC-Sarsa(λ)	20
June 26	Testing NAC-Sarsa(λ)	10
[‡]July 3	n/a	-
July 10	Implementing NAC-Sarsa(λ) with Dyna	25
July 17	Testing NAC-Sarsa(λ) with Dyna	15
[§]July 24	Implementing Q-PAMDP with NAC-Sarsa(λ)	20
July 31	Implementing Q-PAMDP with NAC-Sarsa(λ) and Dyna	20
August 7	Hyperparameter Search	10
August 14	Hyperparameter Search	10
August 21	Experimental Setup	15
August 28	Initial Experiments	20
September 4	Final Experiments	-
[¶]September 11	Final Experiments	-
September 18	Results gathering and analysis	10
September 25	Report write-up	15
October 2	Report write-up	15
October 9	Report write-up	20
October 16	*Final research report due*	-

Table 4.1: Proposed weekly research schedule

4.4 Potential Issues

We note several potential issues that may arise during research, specifically during the implementation stage, Section 4.2.1.

NAC-Sarsa(λ)

Despite our confidence that the NAC-Sarsa(λ) algorithm satisfies the convergence property required for it to be used as a parameter update algorithm in Q-PAMDP, see Section 3.3, the possibility exists that the algorithm simply does not work in practice, even after a hyperparameter search. By which we mean that Q-PAMDP with NAC-Sarsa(λ) either always diverges during learning, almost always gets stuck in a bad local optimum, or does not show any improvement over a baseline agent, i.e. it doesn't learn. If the modified algorithm just converges a bit slower than Q-PAMDP with eNAC or gets a slightly lower average return, the results of this research will still be valid as the modified algorithm may still benefit from Dyna planning updates and may perform better on different domains. On the other hand, if the modified algorithm doesn't work at all, there are several potential fixes to consider, such as changing the function approximation basis, using adaptive learning rates, or changing the reward structure of the domains. Additionally, there are several variants of incremental actor-critic algorithms, such as those presented by Bhatnagar *et al.* [2009], we could try implementing instead. If none of the incremental actor-critic algorithms work, a major change to our

[‡]CHPC Winter School
[§]Second semester begins
[¶]Mid-semester break

model integration approach would be required since the Linear Dyna algorithm we intend to use relies on a linear value function approximator and iterative updates. There are alternative model-based techniques for policy search, such as the Pegasus algorithm Ng *et al.* [2004], that we can consider using.

Model Representation

Linear Dyna uses a linear model representation of the environment's forward transition function, Section 2.3.1. We may find during our testing that this representation is insufficient or inappropriate for the case of parameterised actions on the experiment domains. In this case, we can still apply Dyna-style planning updates, the only changes required would be how the model is represented and learned. For example, Ng *et al.* [2004] learn models using locally weighted linear regression with added Guassian noise. Alternatively, an artificial neural network or decision tree could be used to construct a model offline, although we would then have to take particular care to explore the state-action space sufficiently for the initial model learning. In any case, we would still make use of the model by simulating transitions and performing iterative planning updates.

Computational Requirements

We noted in the time plan that the final experiments could be finished within one week of computational time on the university cluster. In the worst case that the cluster is unavailable at that time, the final experiments will have to be performed sequentially over 10 days. The number of averaged training sessions can be reduced if time becomes an issue. Each complete training session is usually comprised of around 100000 training episodes, we can potentially reduce this number to 80000 if need be without compromising the results. In terms of pre-experimentation testing, the computational requirements are not a major concern. Most of the parameter search, for example, can be performed using half training sessions if necessary as it is quickly apparent whether or not the learning algorithm will diverge or get stuck in a bad local optimum with a particular combination of hyperparameters.

4.5 Conclusion

The proposed research was broken down into three stages and the specific implementation requirements were detailed for each stage. We presented a 4 month time plan that scheduled each step of the research from initial implementation through to final experimentation and report writing. We identified the primary issues that could arise during the course of the research: an inappropriate choice of algorithm or model representation, and general time concerns due to the computational requirements of the experiments, and proposed provisions for such difficulties.

Chapter 5

Conclusion

The recent formulation of parameterised action spaces allows for reinforcement learning algorithms to solve more complex tasks that require a mix of high-level actions and low-level control. Existing techniques for parameterised action spaces, such as the Q-PAMDP algorithm described by Masson *et al.* [2016], are model-free and as such can suffer from slow learning speeds due to poor data efficiency. Model-based algorithms that learn explicit models of the environment can make use of planning, simulating samples from the environment, and have been shown to converge with fewer training episodes than model-free algorithms.

This research aims to extend the Q-PAMDP algorithm with a model-based framework to increase its learning speed. In Chapter 3 we proposed using an incremental natural actor-critic method, NAC-Sarsa(λ), for Q-PAMDP's parameter update step, integrated with Linear Dyna to learn a model online for planning updates. We intend to empirically evaluate the resulting algorithm on the same domains presented by Masson *et al.* [2016], comparing its learning speed against that of model-free Q-PAMDP in terms of the number of episodes taken to converge to a good policy.

Finally, the overall research plan, detailing implementation requirements and time allocations over a 4 month period, was discussed in Chapter 4. It is hoped that this research will further expand the range of control problems that reinforcement learning can solve in a reasonable time, eventually being able to apply to real-world problems that require complex action control.

Bibliography

[Bertsekas 1987] Dimitri P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[Bhatnagar *et al.* 2009] Shalabh Bhatnagar, Richard S. Sutton, Mohammad Ghavamzadeh, and Mark Lee. Natural actor-critic algorithms. *Automatica*, 45(11):2471 – 2482, 2009.

[Boyan 2002] Justin A. Boyan. Technical update: Least-squares temporal difference learning. *Machine Learning*, 49(2):233–246, 2002.

[Bradtke and Barto 1996] Steven J. Bradtke and Andrew G. Barto. Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22(1):33–57, 1996.

[Deisenroth *et al.* 2013] Marc Peter Deisenroth, Gerhard Neumann, and Jan Peters. A survey on policy search for robotics. *Foundations and Trends in Robotics*, 2(12):1–142, 2013.

[Hausknecht and Stone 2016] Matthew Hausknecht and Peter Stone. Deep reinforcement learning in parameterized action space. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[Kakade 2001] Sham Kakade. A natural policy gradient. In *Advances in Neural Information Processing Systems 14*, pages 1531–1538. MIT Press, 2001.

[Khamassi and Tzafestas 2016] Mehdi Khamassi and Costas Tzafestas. Active exploration in parameterized reinforcement learning. *CoRR*, abs/1610.01986, 2016.

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

[Kuvayev and Sutton 1996] Leonid Kuvayev and Richard S. Sutton. Model-based reinforcement learning with an approximate, learned model. In *Proceedings of the Ninth Yale Workshop on Adaptive and Learning Systems*, pages 101–105, 1996.

[Masson *et al.* 2016] Warwick Masson, Pravesh Ranchod, and George Konidaris. Reinforcement learning with parameterized actions. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 1934–1940. AAAI Press, 2016.

[Ng *et al.* 2004] Andrew Y. Ng, H. J. Kim, Michael I. Jordan, and Shankar Sastry. Autonomous helicopter flight via reinforcement learning. In *Advances in Neural Information Processing Systems 16*, pages 799–806. MIT Press, 2004.

[Parr *et al.* 2008] Ronald Parr, Lihong Li, Gavin Taylor, Christopher Painter-Wakefield, and Michael L. Littman. An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML'08, pages 752–759, New York, NY, USA, 2008. ACM.

[Pazis and Lagoudakis 2009] Jason Pazis and Michail G. Lagoudakis. Binary action search for learning continuous-action control policies. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 793–800, New York, NY, USA, 2009. ACM.

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

[Sutton *et al.* 2008] Richard S. Sutton, Csaba Szepesvári, Alborz Geramifard, and Michael Bowling. Dyna-style planning with linear function approximation and prioritized sweeping. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, UAI'08, pages 528–536, Arlington, Virginia, United States, 2008. AUAI Press.

[Sutton 1990] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224. Morgan Kaufmann, 1990.

[Thomas 2014] Philip Thomas. Bias in natural actor-critic algorithms. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 441–448, Bejing, China, 2014. PMLR.

[Watkins and Dayan 1992] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.