Thesis Proposal Review: "Reinforcement Learning for Large Continuous Spaces"

Matthew Robards

November 9, 2009

Contents

1 Introduction

This project looks at scalability issues in reinforcement learning (RL) in large state and action spaces. This is motivated by the inability of RL to handle such large spaces in real world applications, and in particular, interactive game AI (which I later claim to be the perfect real world RL test bed). In particular it will look at reinforcement learning in *continuous* state/action spaces.

In the literature review I introduce what I consider "traditional" reinforcement learning algorithms. That is, those algorithms discussed in Sutton and Barto [29] (considered here the "Bible of RL". I will introduce these methods in the coherent structure used in the afore-mentioned. Specifically I will firstly introduce the RL problem, and introduce each method firstly in the context of prediction (also known as policy evaluation, where an agent forms beliefs about its policy) and then introduce the notion of control (in which an agent selects a policy based on its beliefs).

I will then continue to discuss the more modern algorithms in the context of various distinguishing features. Specifically I distinguish between those methods which use function approximation, those which approximate the underlying process, and the ways in which they deal with continuous spaces.

Part I
The Old Testament

2 k-Armed Bandit

The k-armed bandit problem considers the dilemma of choosing actions $a_t \in \mathcal{A}$, from a finite action space, at time t repeatedly in such a way that the total reward $R_0 = \sum_{t=0}^{T} r(a_t)$ (using the notation that $R_t = \sum_{i=t}^{T} r(a_t)$) (for $r(a_t) = r_t \in \mathbb{R}$ the immediate reward recieved for choosing action a_t) is maximized. The more illustrative example of k slot machines is typically given [29]. That is, at time t the agent has the choice of pulling the arm on one of k slot machines, which either pay off $r_t = 1$ or don't $r_t = 0$. Each slot machine pays a reward $r_t \in \{0, 1\}$, with each machine having a different payoff probability distribution. The aim for the agent is to, given a budget of T pulls, maximize the return R_T by discovering which arm pays off most frequently and selecting the corresponding action. Note here the distinction between reward r_t – the instantaneous reward – and return R_t – the total return after t discrete time steps.

The k-armed bandit was first introduced by Robbins in [25] through the question of random sampling from different populations. Formally, he asks "A problem of two populations" [25]: Given two populations A and B, with unknown cumulative distribution functions $F(x)$ and $G(x)$ respectively, and expectations α and β respectively given by

$$\alpha = \int_{-\infty}^{\infty} x dF(x) \tag{1}$$

$$\beta = \int_{-\infty}^{\infty} x dG(x) \tag{2}$$

how can $x_1, x_2, \ldots, x_n$ be sampled from A and B such that the expected value of the sum $S_n = \sum_{i=1}^{n} x_i$ is maximized.

Robbins then puts forward the simple example of flipping two coins, each with (possibly different) unknown bias. If a coin comes up heads, a payoff of 1 is achieved, and 0 for tails. Hence A and B represent the coins, and α and β represent the probabilities of tossing heads on the two coins respectively. He claims that the intuitive agent would sample with bias towards A or B once one of the distributions was perceived to pay better than the other.

Now let us define some terminology which will carry us through the entire study of reinforcement learning (RL). This nomenclature, which is now common amongst the RL community, will be borrowed from [29]. Firstly the estimated *value* $Q(a_t)$ of an action a_t at time t refers to the agents current belief of the reward r_t which will be received if action a_t is followed. One estimate of the value of an action for the k-armed bandit problem is given by the expectation

$$Q(a_t) = \mathbb{E}\{r_t\} = \frac{1}{n_t} \sum_{i=\{0,\ldots,t\}:a_i=a_t} r_i \tag{3}$$

where $n_t = \sum_{i=0}^{t} \delta_{a_t,a_i}$ is the number of times action a_t has been selected, $\delta_{a,b}$ is the Kronecker delta indicator.

Given a value estimate for each action, there is at least one action whose value will dominate. This is called the *greedy* action. More formally the greedy action, at time t, is given by

$$a_t = \underset{a}{\operatorname{argmax}}\, Q(a). \tag{4}$$

This notion of greedy actions gives rise to one of the fundamental problems in the field of RL. This problem, known as *exploration vs. exploitation* regards the choice of exploitative greedy actions, or exploratory actions. That is, given value estimates the agent must ask "do I exploit the fact that I know this arm pays reasonably well, or do I explore in hope that I will find an arm that pays even better?" Robbins' agent who samples from the coins with bias toward the coin percieved to pay best must also consider that further trials from the "worse" coin may cause him to rethink his initial hypothesis. Although seemingly simple, this problem is very deep and widely studied. For example, given a budget of only T pulls it may waste valuable time to explore, however, it may also be found that a far better action can be taken than that which is currently considered optimal. As stated by [10], this problem asks the bigger question of "How do we devise an algorithm that will efficiently find the optimal value function?" For the 2-armed bandit, the approach of switching arms when the current one does not pay off performs better than random arm selection [25], however it is also known that this is not an optimal solution [11].

2.1 Prediction

In RL, prediction refers to the problem of finding (action) values Q for actions a which best estimate the true action value $Q^*(a)$. That is, prediction attempts to "devise an algorithm that will efficiently find the optimal value function?" [10]. As previously stated, perhaps the simplest value estimate is given by the expectation in equation (3) (taken from [29]). More sophisticated value estimates will be used in later discussed algorithms, however for this simple problem only the average reward will be considered. The expected average reward can be written incrementally as in [29]:

$$Q(a_{t+1}) = Q(a_t) + \frac{1}{t+1}\left(r_{t+1} - Q(a_t)\right) \tag{5}$$

This incremental approach gives the tools to perform two more sophisticated tasks. Firstly, one can create a weighted average by replacing $\frac{1}{t+1}$ with a step-size α. The incremental update then reduces to

$$\begin{align} Q(a_{t+1}) &= Q(a_t) + \alpha\left(r_{t+1} - Q(a_t)\right) \tag{6}\\ &= (1-\alpha)^t Q(a_0) + \sum_{i=0}^{t} \alpha(1-\alpha)^{t-i} r_i. \tag{7}\end{align}$$

Typically the incremental update in equation (6) is used in practice. (7) however shows that this is a weighted average since the weights all sum to 1:

$$(1-\alpha)^t + \sum_{i=1}^{t} \alpha(1-\alpha)^{t-i} = 1, \forall t \in \mathbb{N}; t \geq 1.$$

This can easily be proven by induction [29]. Now the weight on rewards ($\alpha(1-\alpha)^{t-i}$) diminishes greatly for rewards recieved long in the past. This approach then is good for "Tracking a Nonstationary Problem" [29]. That is, if the reward distributions are changing for each bandit, this new incremental update is best since it places highest weight on most recent rewards. Therefore for a slowly changing distribution, the older action values are gradually forgotten.

Further, we can still guarantee convergence by enforcing the Robbins-Monroe conditions on α:

$$\sum_{t=1}^{\infty} \alpha_t(a) = \infty \tag{8}$$

and,

$$\sum_{t=1}^{\infty} \alpha_t^2(a) < \infty \tag{9}$$

where $\alpha_t(a)$ is the step size after the $(t-1)$th selection of action a.

The second useful trick which utilizes the incremental update is an exploration strategy called *UCB*. The problem of exploration versus exploitation is discussed in the next section.

2.2 Action Seletction And Exploration

The UCB exploration strategy for exploration has proven finite regret bounds on the k-armed bandit [2]. The algorithm simply plays an arm based on a function of the upper confidence bound on the mean estimate, which is calculated based on standard error measures using how frequently the arm has previously been played. That is, an arm which has rarely been played will likely have a higher standard error, and be preferred over one which has been played extensively and has a tighter standard error. This is similar to the optimistic initial values approach which initializes the values excessively. As the agent then gains experience, the action values corresponding to the most used actions will "become more exact, and therefore, lower" [31]. Those actions not yet used, however, will still have an excessive value and hence the agent will desire those more. These methods encourage the agent to initially choose actions uniformly. Optimistic initial values (or exploring starts [31]), presented in [29] has been proven to converge to the optimal in polynomial time given sufficiently high initial values [7]. Perhaps a more interesting problem is that of control. The control problem asks given our belief about the value of an action, how do we choose our next action? This may seem simple – choose the greedy action $a_t = \text{argmax}_a\, Q(a)$ – however one must keep in mind the exploration vs. exploitation problem. That is, only taking the greedy action means 100%-0% exploitation-exploration. A straight forward alternative is ϵ-greedy action selection [29] in which the greedy action a_t is chosen with probability $1-\epsilon$, and with probability ϵ a random action is taken. This control method proved to be quite useful in [29] on the so called 10-armed test bed. In this task 2000 random 10-armed bandit tasks were generated. Each task was run for 1000 episodes and the rewards for each action a were sampled from a gaussian distribution with mean $Q^*(a)$ and variance 1. Figure (1) shows the results the authors obtained.

The authors further point out one shortcoming of ϵ-greedy control is that there is no discrimination between exploratory moves. They further propose

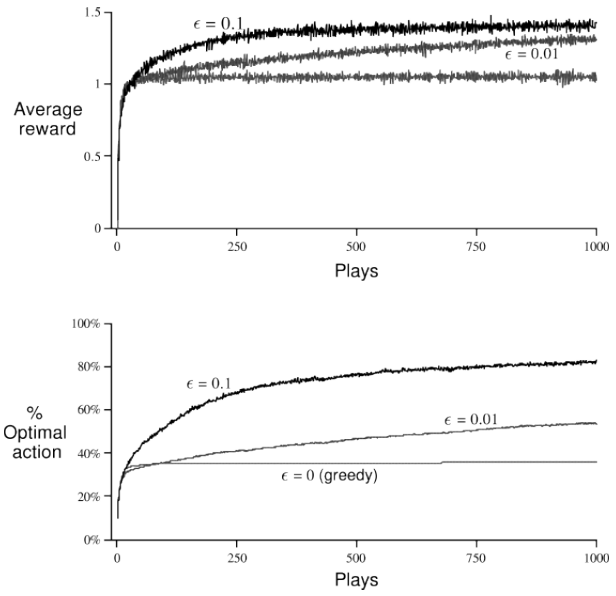


Figure 1: Performance of ϵ-greedy action selection using average rewards in prediction on the 10-armed testbed. (From [29].)

softmax, or Boltzmann action selection. Here actions are sampled randomly according to the current estimate of their values. Action a is chosen at time t with probability

$$P(a) = \frac{e^{Q_t(a)/\tau}}{\sum_{a'} e^{Q_t(a')/\tau}} \tag{10}$$

where τ is called the temperature. Now actions are selected randomly with probability based on the current belief about the actions.

3 Reinforcement Learning

3.1 The Reinforcement Learning Model

In the previous section, the problem of the k-armed bandit was addressed in which the agent selects actions $a \in \mathcal{A}$ in an attempt to maximize the total reward recieved $R_0 = \sum_{t=0}^{T} r(a_t)$. Now consider the bigger RL problem. This problem is a setup in which the agent at time t observes its state $s_t \in \mathcal{S}$ in the environment, chooses its action $a_t \in \mathcal{A}$ and receives a (possibly) stochastic reward $r_t = R(s_t, a_t, s_{t+1})$ [10, 12, 29]. This constitutes the reinforcement learning model, in contrast to the bandit problem, in which the agent always remains in the same state. Traditionally the sets $\mathcal{A}$ an $\mathcal{S}$ are finite, however we will later consider the case of infinite state and action spaces with more modern algorithms.

Given this RL model, the agent attempts to behave optimally, where optimality can be defined by different objective criteria[12]. Common models for optimal behavior, given in [12], include the finite horizon model

$$R = \mathbb{E}\left(\sum_{t=0}^{h} r_t\right), \tag{11}$$

the infinite horizon discounted model

$$R = \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t r_t\right), 0 \leq \gamma < 1, \tag{12}$$

and the long run average reward

$$R = \lim_{h\to\infty} \mathbb{E}\left(\frac{1}{h}\sum_{t=0}^{h} r_t\right), \tag{13}$$

where h is called the horizon. In the finite horizon model, the agent must maximize the expected reward over the next h steps without concern for what happens later than that [12].

The infinite horizon model in contrast considers all rewards into the future. Here the discount factor γ^t which the authors of [12] consider to be an interest rate (in that it diminishes the return over time), or probability of living another step, ensures that the expected return is bounded. This discount has the effect of making the agent regard very distant future rewards as less desirable than those which can be obtained in the near future. This optimality model has been very widely studied [12], and is the basis for many current state of the art algorithms. Note that a discounted MDP can be transformed to an undiscounted one by simply taking the "probability of termination" γ into account in the state transition function.

The long run average reward model can be seen as the limiting case of the infinite discounted model as $\gamma \to 1$ [12]. This model has the drawback, however, that there is no distinguishing between solutions which obtain rewards early in the simulation or later. That is, assuming a reward is achieved, there is no discrimination based on how far into the future it was received.

In choosing an objective criteria, it is important to consider the nature of the task. That is, consider if the task is *episodic* or not [29]. An episodic task is one which terminates at some time, like a game of checkers. Non-episodic tasks continue indefinitely, like a robot acting in everyday life. Obviously the finite horizon model would be suitable for finite tasks. The infinite horizon model however requires that the task continue to infinity. This is dealt with in episodic tasks by considering termination as entering an absorbing state in which nothing happens until infinity.

Example applications for the reinforcement model include board games tic-tac-toe [8, 17], Go [5, 8, 17], Chess [5, 8, 17] and backgammon[8, 32]. In these games, a natural reward would be ± 1 corresponding to win or loss respectively [29]. Note then that the reward is not received until after the game is complete. This is called delayed reward, and is necessary in such situations as the agent will often be unsure after a specific state action pair if his action was good or not. Consider for example a chess move which is perceived to be good, but plays into the opponents trick. This perceptually good move is then in fact a bad move.

Further examples of the reinforcement learning model include robotics. Take as an example the recycling robot [29]. This robot must collect empty cans from the office environment, and place them in a receptacle. Positive rewards are achieved for collecting cans, and negative reward is given for running out of battery (at which stage a human must recharge the robot). This robot must then collect as many cans as poosible, but always return to a recharge station when low on battery.

3.2 The Markov Decision Process

The next thing that must be defined to understand RL is the Markov decision process (MDP). [29] defines a state signal to be *Markov* or have the *Markov Property* if it retains all relevant information about the past. The authors give as an example the flight of a cannonball. That is, given the current position and (directed) velocity of the ball, we know everything necessary to estimate the next position of the ball(neglecting wind etc.). "If this information is available for the current time, then it is not necessary to know anything more about the past history of the process" [35]. The MDP is an environment $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$ within which the Markov property holds on the state signal.

More formally, the dynamics of a Markov state signal can be defined by the probability distribution

$$Pr\left(s_{t+1} = s', r_{t+1} = r | s_t, a_t\right), \tag{14}$$

as opposed to a non-Markov system which requires more prior information

$$Pr\left(s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\right). \tag{15}$$

The Markov property defines (14 and 15) to be equivelant. The transition probability is defined [12, 23] as

$$T(s, a, s') = Pr\left(s_{t+1} = s' | s_t = s, a_t = a\right), \tag{16}$$

and the expected reward

$$R(s, a, s') = \mathbb{E}\left(r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\right) \tag{17}$$

Further, by marginalizing over s' we can define

$$R(s, a) = \sum_{s'} T(s, a, s') R(s, a, s') \tag{18}$$

The goal of the intelligent agent in the MDP is to find a policy (set of conditional rules either deterministic or probabilistic) that will maximize the agents positive sensation, or minimize some cost [23]. The author describes this policy, denoted by π, as a scheme for choosing actions $a \in \mathbb{A}$ at states $s \in \mathbb{S}$. More formally a stochastic policy π is a function $\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a function mapping states and actions to probabilities. A deterministic policy is defined by $\pi(s, a) \in \{0, 1\} \forall s, a$ The agent may consider either stationary or non-stationary policies. Stationary policies are unchanged mappings from states to actions, whereas non-stationary policies can change between, for example, various iterations of simulations. Now we can consider the optimal policy π^*

to be that which maximizes a given optimality criteria [23]. For example, the optimal policy based on the infinite discounted model for optimality (12) is given by

$$\pi^* = \operatorname*{argmax}_{\pi} \sum_{t=0}^{\infty} V^{\pi}(s_t) \tag{19}$$

3.3 Value Functions

Common to most, if not all, reinforcement learning algorithms [12, 23, 29, 31] are value functions. Value functions predict the expected return in a given state, or for a state action pair. A distinction is commonly drawn between these two. Namely the state value function, denoted by $V : \mathbb{S} \to \mathbb{R}$ represents the return the agent expects to see, given he is in state s. Further, denote by $V^{\pi}(s)$ the expected return starting in state s and following a policy π. This is put more formally in [29] as

$$V^{\pi}(s) = \mathbb{E}_{\pi}(R_t | s_t = s) \tag{20}$$

which can be written as follows if using an infinite discounted model of optimality

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s\right). \tag{21}$$

The state-action value, commonly denoted by $Q : \mathbb{S} \times \mathbb{A} \to \mathbb{R}$ represents the expected future return if the agent takes action $a \in \mathbb{A}$ when in state $s \in \mathbb{S}$. Further $Q^{\pi}(s, a)$ is the value of taking action a when in state s, and following policy π thereafter. This is put more formally as

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}(R_t | s_t = s, a_t = a). \tag{22}$$

Given these notations of the value functions we can now define optimal values and policies. An optimal state value function is given in [29] as

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{23}$$

and the optimal state value function is given by

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a). \tag{24}$$

Moreover, we can now define the optimal policy π^* as

$$\begin{aligned} \pi^*(s) &= \operatorname*{argmax}_{a} Q(s, a) \\ &= \operatorname*{argmax}_{a} \mathbb{E}_{\pi}(r_t | s_t = s, a_t = a), \forall s \end{aligned} \tag{25}$$

which is evidently equivelant to (19) for the discounted infinite model of optimality.

The following sections consider algorithms to solve the problem of how to find, or best approximate, this optimal policy.

4 “Traditional” Model Based Reinforcement Learning Algorithms

Model based approaches use dynamic programming to, given a (perfect) model of the environment, compute an optimal policy [29]. Such a model is difficult, if not impossible, to obtain for most environments, however such methods are still an important foundation for the remainder of RL work.

4.1 Dynamic Programming

I assume a finite MDP with probability of transition from state s to s' through action a given by $T(s, a, s')$ and reward function $R(s, a, s^prime)$ defined as before.

Dynamic programming then directly finds the optimal value function satisfying the Bellman equations [29]:

$$\begin{aligned} V^*(s) &= \max_a \mathbb{E}(r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a) \\ &= \max_a \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V^*(s_{t+1})) \end{aligned} \tag{26}$$

4.2 Policy Evaluation

In policy evaluation [29], the state value function is evaluated for a policy π. This contrasts equation (26) which finds the value function under the optimal policy. Hence, instead of the state value function being taken as the maximum over all possible actions, it is taken as the expectation over actions.

$$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V^\pi(s_{t+1})) \tag{27}$$

4.3 Policy Iteration

Policy iteration is a two step iterative procedure in which firstly the policy V^π is evaluated, and the the policy π is updated. V^π is updated through policy evaluation as described above. π is then updated to π' through

$$\pi'(s) = \operatorname*{argmax}_a Q^\pi(s, a) \tag{28}$$

where $Q^\pi(s, a)$ is defined to be the value of following action a in state s, and π thereafter:

$$\begin{aligned} Q^\pi(s, a) &= \mathbb{E}(r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s, a_t = a) \\ &= \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V^\pi(s_{t+1})) \end{aligned} \tag{29}$$

The policy π under policy iteration converges to the optimal π^*.

4.4 Value Iteration

Value iteration is a procedure which combines the two stages of policy iteration into a one step iterative procedure [29]. The new value function update is now given by:

$$\begin{aligned} V_{k+1}(s) &= \max_a \mathbb{E}(r_{t+1} + \gamma V_k(s_{t+1}) | s_t = s, a_t = a) \\ &= \max_a \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V_k(s_{t+1})) \end{aligned} \tag{30}$$

5 "Traditional" Model Free Reinforcement Learning Algorithms

5.1 Monte-Carlo Reinforcement Learning

5.1.1 Prediction

Monte-Carlo (MC) RL simply estimates the expected return from a state $s \in S$ as the average of the rewards experienced after s [29]. That is, "MC algorithms treat the long-term reward as a random variable and take as its estimate the sampled mean" [34]. More formally,

$$V^\pi(s) = \mathbb{E}(R_t | s_t = s) = \mathbb{E}\left(\sum_{i=t}^{T} \gamma^i r_i\right), \tag{31}$$

where T is the termination of the episode. Hence, we naturally require an episodic task ($T < \infty$) for MC algorithms. It holds that the average should converge to the expected value as the number of observed rewards is increased [29].

It is essential for the control problem to further define the state-action value. This is simply the analog to the state values, and is given by

$$Q^\pi(s,a) = \mathbb{E}(R_t | s_t = s, a_t = a) = \mathbb{E}\left(\sum_{i=t}^{T} \gamma^i r_i\right). \tag{32}$$

The Monte-Carlo learner attempts to estimate these values through an update function of the form

$$V_{t+1}(s) = V_t(s) + \alpha\left(R_t - V_t(s)\right) \tag{33}$$

or for action values

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\left(R_t - Q_t(s,a)\right). \tag{34}$$

In other words the estimated value, $V_t(s)$ or $Q_t(s)$, is updated at each time step according to the error $R_t - V_t(s)$.

5.1.2 Control

Monte-Carlo control is easily performed using generalized policy iteration [29]. This is a two step procedure in which the state-action values for a policy are improved, and the policy is subsequently updated based on these new values. That is, the state-action values are updated for a state through equation (32), then the policy is updated through equation (25). This process is shown in Figure (2).

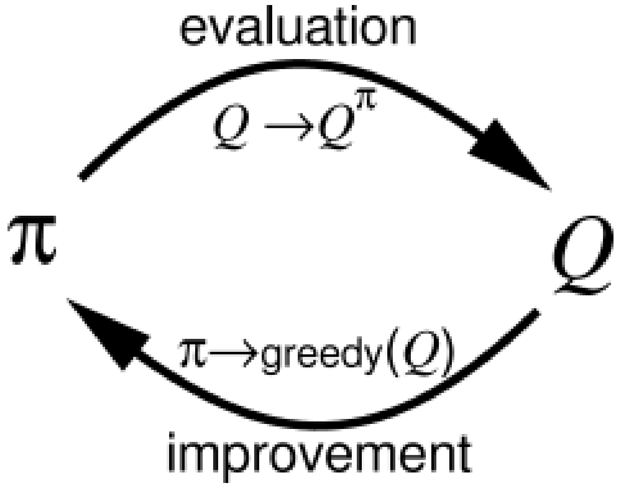


Figure 2: Generalized policy improvement for the Mote-Carlo algorithm. (From [29].)

5.2 Temporal Difference Reinforcement Learning

Temporal difference (TD) learning, like Monte-Carlo learning, bases its estimates V of the value V^{π} of a policy on experience [29]. TD methods, unlike Monte-Carlo methods, however, are not required to await the termination of an episode to update the value functions. In fact, TD methods need only wait until the next time step.

5.2.1 TD(0) – Prediction

The simplest form of TD learning, TD(0), regards the return R_t for an agent currently in state s_t to be the sum of the instantaneous reward following s_t, and the return thereafter. Mathematically (from [29]):

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{\pi}(R_t | s_t = s) \\ &= \mathbb{E}_{\pi}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right) \\ &= \mathbb{E}_{\pi}\left(r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \middle| s_t = s \right) \\ &= \mathbb{E}_{\pi}\left(r_{t+1} + \gamma V^{\pi}(s_{t+1}) \right). \end{aligned} \tag{35}$$

Here t may be finite (episodic task), in which case one should choose $\gamma = 1$, or alternatively when $t = \infty$, $0 < \gamma < 1$ should be chosen.

Given this expected return, the TD agent attempts to estimate it with V. Recall the update equations

$$V_{t+1}(s) = V_t(s) + \alpha \left(R_t - V_t(s) \right). \tag{36}$$

Now using the TD(0) return, the agent updates its estimate of the values through

$$V_{t+1}(s_t) = V_t(s_t) + \alpha \left[r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \right]. \tag{37}$$

This method, through its use of previous value estimates in update, is known as a bootstrapping method. "They learn a guess from a guess" [29]. Further TD(0) has been proven to converge for a fixed π to the optimal V^{π}.

5.2.2 SARSA – On-Policy TD Control

The algorithm Sarsa, so named for its state $\rightarrow$ action $\rightarrow$ reward $\rightarrow$ state $\rightarrow$ action cycle, is the on-policy control version of TD learning. This algorithm follows generalized policy iteration by first estimating the action value function, and then updating the policy accordingly. More specifically Sarsa, at each time step, updates the action value as in [29] through

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \right]$$

and further updates the policy based on

$$\pi(s) = \operatorname*{argmax}_a Q(s, a).$$

Obviously this policy iteration equation could change to accommodate exploration. For example, we could use the ϵ-greedy update as in [29] in which the greedy action is then chosen with ϵ probability, and a random action is chosen with probability $1 - \epsilon$.

5.2.3 Q-Learning – Off-Policy TD Control

According to [29] Q-learning, which is the off policy version of TD control introduced by [35], is one of the most significant breakthroughs in reinforcement learning. The simple one-step version of Q-learning updates the action values through

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t) \right].$$

This updates the estimate of the action value based on the error $r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)$, where Q_t is used to estimate the optimal value Q^* independently of the action being taken. Here the one step return, which is given by $R_t = r_{t+1} + \gamma \max_a Q^*(s_{t+1}, a)$, is the next received reward plus the estimated value of all subsequent rewards. Then the update is based on exactly the difference between the estimated action value for action a_t, and the return for taking action a_t and following the optimal policy thereafter.

As with all methods, the policy update can then be performed using a greedy policy, as in (25), or alternatively exploratory actions can be taken using such methods as ϵ-greedy policy update.

5.2.4 TD(λ) – Interpolating Between TD and Monte-Carlo

Several model free methods have now been introduced in terms of the update formula

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha Err(s,a;t) \tag{38}$$

where $Err(s,a;t)$ is the error between the estimated value of taking action a in state s, and the so-called "true" return at time t. Generally the error is given by

$$Err(s,a;t) = R_t - Q_t(s,a) \tag{39}$$

where R_t is the return at t defined in various ways. eg. long term average reward, etc.

Now we will look at TD(λ) which attempts to interpolate between the Monte-Carlo return

$$R_t = r_t + \gamma r_{t+1} + \ldots + \gamma^{T-t-1} r_T \tag{40}$$

and the one step TD(0) return given

$$R_t^{(1)} = r_t + \gamma Q(s_{t+1}, a_{t+1}). \tag{41}$$

Firstly, define the n step return [29] as

$$R_t^{(n)} = r_t + \gamma r_{t+1} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n Q_t(s_{t+n}, a_{t+n}). \tag{42}$$

Now TD(λ) takes a weighted average of the 1-step, 2-step, ..., ∞-step return, putting more importance on the shorter returns. More formally the TD(λ) return, as in [29], is given by

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(\lambda)}. \tag{43}$$

6 Reinforcement Learning With Function Approximation

The value representation discussed thus far are called tabular representations. This is because the value $Q(s,a)$ is stored in a table indexed by s, and a. This representation, although convenient, is very expensive in terms of computation and storage [29]. Also it does not generalize. That is, given a state s, a similar state s', and an action a, a good estimate of $Q(s,a)$ tells us nothing about $Q(s',a)$. Function approximation solves both these issues [29].

Where $Q(s,a)$ was previously represented tabularly, we now represent it as $Q(s,a;w) = \langle w, \phi(s,a)\rangle$, where $\phi(s,a)$ is some feature map describing the state action pair. Just as before where the Q value of a state action pair was updated based on the error, now the Q function is updated through its parameters based on the error. One typical update formula from [29] is the gradient descent update given by

$$w_{t+1} = w_t + \frac{\alpha}{2}\nabla_{w_t} Loss(R_t, s_t, a_t), \tag{44}$$

where $Loss(R_t, s_t, a_t)$ can be computed according to various loss functions. In [29], the authors use the L_2 loss given by $Loss(R_t, s_t, a_t) = Err^2(R_t, s_t, a_t) = (R_t - \langle w, \phi(s_t, a_t)\rangle)^2$. Note here that a linear function approximator has been adopted. Hence the update equation becomes

$$w_{t+1} = w_t - \alpha Err(s_t, a_t) \cdot \phi(s_t, a_t), \tag{45}$$

where $a \cdot \vec{b}$ is defined here as scalar vector multiplication. As in previous sections the return R_t can be defined in many ways (eg, Monte-Carlo return, TD return, etc.) with different models of optimality (eg. infinite discounted and finite return). It should be noted here though that, as stated in [29], RL with function approximation is only guaranteed to converge to the optimal value function under the Robbins-Munroe stochastic approximation conditions, and when R_t is an unbiased estimate of the true value $Q^\pi(s_t, a_t)$ under policy π. Hence Monte-Carlo RL is guaranteed to converge, as is TD(λ) for $\lambda = 1$, according to [29]. Nonetheless, other methods are still used with function approximation since they still often converge to some "good" local optimum in practice.

6.1 Major Issues

Numerous algorithms have been proposed to address the convergence issues of such algorithms as TD(λ), including VAPS [3], which is a generalized modification which can be made to many existing algorithms to guarantee convergence.

A further issue with function approximation is that using function approximation with bootstrapping methods such as Q-learning is highly dangerous due to the susceptibility to accumulating errors. It has been claimed that value function approximators are prone to over estimation, and are really only valid for making predictions in those regions of the input space which are well covered by training samples[28]. This is problematic since small errors can accumulate through bootstrapping, and eventually result in quite large errors.

This problem is addressed by an algorithm coined HEDGER in [28]. This algorithm alleviates the problems by only considering input samples $\vec{x}$ which lie in the convex hull

$$\vec{x}^T (X^T X)^{-1} \vec{x} \leq max_i v_{ii} \tag{46}$$

where v_{ii} are the diagonal elements of the hat matrix given by

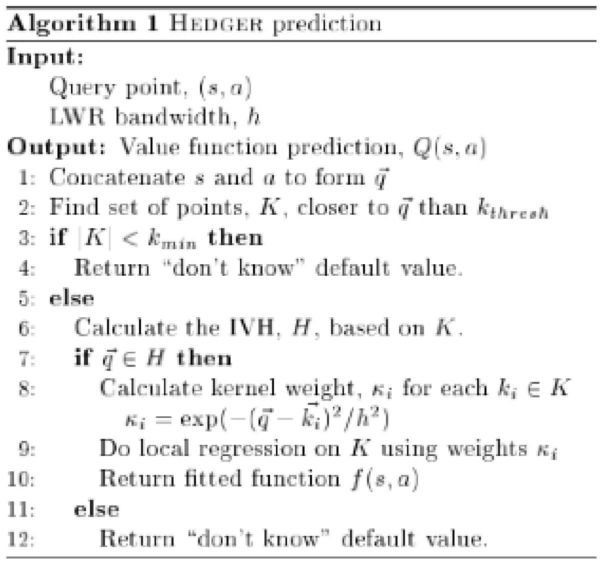
$$V = X(X^T X)^{-1} X^T \tag{47}$$

where the rows of X are the input samples. The algorithms is then broken into a training phase and a prediction phase as discussed later.

A further issue addressed by [28] is that of initialization. More specifically, in real-world/large problems, the agent has a very large state-action space to explore before finding "good" examples. This problem is addressed by providing the system with some initial example episodes which are known to perform well. The agent then learns to imitate these behaviors before moving on to further learning independently.

In prediction the algorithm, for each input vector $\vec{x}$, finds a set of points K within distance k_{thresh} of $\vec{x}$. It is claimed that $|K|$ (the cardinality of K) must be at least the number of parameters the regression attempts to fit. Hence, k_{min}

is set, typically to be the number of parameters. If $|K| < k_{\text{min}}$ a default value $Q =$"dont know" is returned. Otherwise, assuming the input vector $\vec{x}$ lies in the convex hull 46, locally weighted regression is performed, where the weights are based on the input sample's distance from the training samples.



Algorithm 1 HEDGER prediction

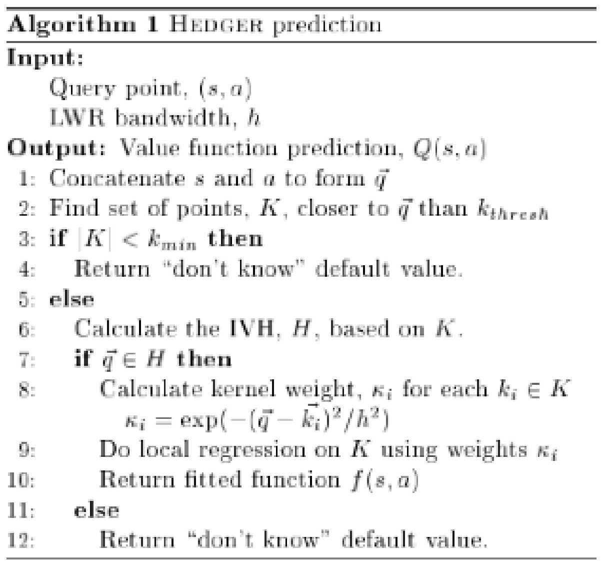
```
Input:
    Query point, $(s, a)$
    LWR bandwidth, $h$
Output: Value function prediction, $Q(s, a)$
 1: Concatenate $s$ and $a$ to form $\vec{q}$
 2: Find set of points, $K$, closer to $\vec{q}$ than $k_{thresh}$
 3: if $|K| < k_{min}$ then
 4:     Return "don't know" default value.
 5: else
 6:     Calculate the IVH, $H$, based on $K$.
 7:     if $\vec{q} \in H$ then
 8:         Calculate kernel weight, $\kappa_i$ for each $k_i \in K$
                $\kappa_i = \exp(-(\vec{q} - \vec{k_i})^2/h^2)$
 9:         Do local regression on $K$ using weights $\kappa_i$
10:         Return fitted function $f(s, a)$
11:     else
12:         Return "don't know" default value.
```

Figure 3: HEDGER prediction algorithm taken from [28]

In training, given a state action pair (s_t, a_t), next state s_{t+1} and reward r_t, the algorithm simply performs q-learning. That is, the state action value is found using the prediction algorithm through $q = Q_{\text{predict}}(s, a)$, and the best next action value is found through $q' = max_{a_{t+1}} Q_{\text{predict}}(s_{t+1}, a_{t+1})$. Now $q_{\text{new}} = q + \alpha(r_t + \gamma q_{\text{next}} - q)$ is calculated, and the action values are updated through

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \kappa_i(q_{\text{new}} - Q(s_i, a_I i)), \tag{48}$$

where κ_i are the kernel weights found in the prediction algorithm.



```
Algorithm 1 HEDGER prediction
Input:
    Query point, $(s, a)$
    LWR bandwidth, $h$
Output: Value function prediction, $Q(s, a)$
 1: Concatenate $s$ and $a$ to form $\vec{q}$
 2: Find set of points, $K$, closer to $\vec{q}$ than $k_{thresh}$
 3: if $|K| < k_{min}$ then
 4:   Return "don't know" default value.
 5: else
 6:   Calculate the IVH, $H$, based on $K$.
 7:   if $\vec{q} \in H$ then
 8:     Calculate kernel weight, $\kappa_i$ for each $k_i \in K$
          $\kappa_i = \exp(-(\vec{q} - \vec{k_i})^2 / h^2)$
 9:     Do local regression on $K$ using weights $\kappa_i$
10:     Return fitted function $f(s, a)$
11:   else
12:     Return "don't know" default value.
```

Figure 4: HEDGER training algorithm taken from [28]

Part II
The New Testament

Thus far, an introduction of the methods discussed in [29] has been given. Hereafter a summary of the more recent methods will be given, with significant focus on those algorithms which have been presented to handle continuous states, actions or time, or those which could become powerful when extended to such spaces. These algorithms will be introduced in three categories. Namely, those which approximate the value function, those which approximate the underlying model, and those which deal with continuous spaces (and how).

7 Approximating the Value Function

7.1 Gradient Free Methods

In this section I discuss algorithms which solve the MDP without using (inefficient) gradient based methods to estimate state(-action) values. To this end, much attention has been directed towards fitting trajectories $D = \{s_i, a_i, r_i\}, i = 1, 2, \ldots, N$ of input data. Two such methods, named least squares temporal difference learning (LSTD) [6] and least squares policy iteration (LSPI) [13], use least squares regression to fit the data. These temporal difference algorithms analytically calculate the parameters w which minimize a least squares error, rather than using gradient descent.

The LSTD algorithm represents the large batch TD(λ) return as

$$\begin{aligned} Err(R_t, s_t, a_t) &= \sum_{i=1}^{t} e_i (R_i - Q(s_i, a_i)) && (49) \\ &= \sum_{i=1}^{t} e_i (r_i + Q(s_{t+i}, a_{i+1}) - Q(s_i, a_i)) && (50) \\ &= \sum_{i=1}^{t} e_i \left(r_i + \langle w, [\phi(s_{i+1}, a_{i+1}) - \phi(s_i, a_i)] \rangle\right) && (51) \\ &= \mathbf{b} - w^T \mathbf{A} && (52) \end{aligned}$$

where

$$\begin{aligned} \mathbf{b} &= \sum_{i=1}^{t} e_i r_i && (53) \\ \mathbf{A} &= \sum_{i=1}^{t} e_i \left[\phi(s_i, a_i) - \phi(s_{i+1}, a_{i+1})\right]^T && (54) \end{aligned}$$

and the eligibility trace e_i, given by

$$e_i = \sum_{i=t_0}^{t} \lambda^{i-1} \phi(s_i, a_i), \qquad (55)$$

is the eligibility of (s_i, a_i) to be updated. To illustrate this, consider monte-carlo learning ($\lambda = 1$), where all states in the trajectory are eligible for update.

Once an update of w is required, we can then simply solve

$$w = \mathbf{A}^{-1} \mathbf{b} \qquad (56)$$

which solves $Err(R_t, s_t, a - t) = 0$.

It has been pointed out, however, out that LSTD is unsuitable for policy iteration due to the fact that the error is weighted by the state visitation frequencies (dictated by the current policy) [13]. Further problematic is the fact that LSTD only estimates values for the current policy $Q(s, \pi(s))$, whereas policy iteration requires a full understanding over all policies in order to calculate $\operatorname{argmax}_a Q(s, a)$. In overcoming problems involved with using LSTD for policy iteration, a new algorithm called least squares Q (LSQ) was introduced [13], which safely calculates the state-action values allowing for policy iteration in an algorithm called least squares policy iteration (LSPI) [13].

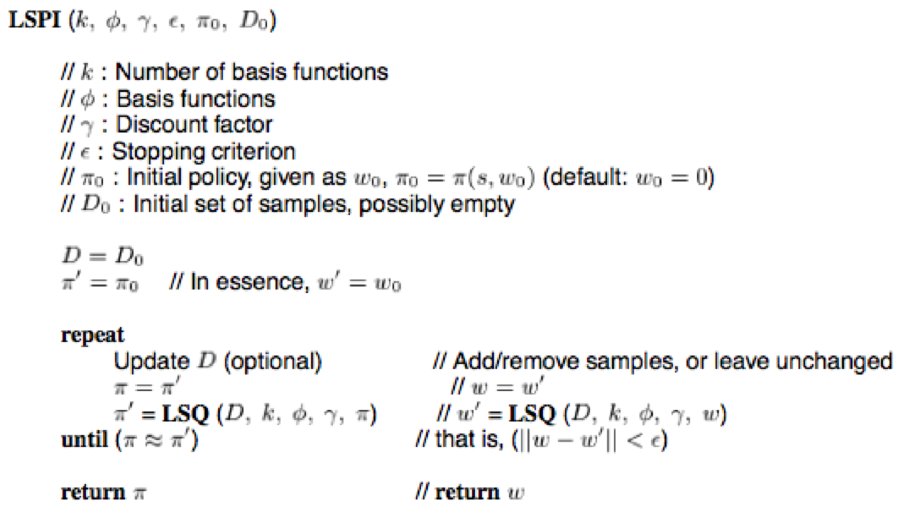
LSQ attempts to analytically calculate the parameters w^π to function $Q^\pi(s, a; w) = w^\pi \phi(s, a)$ which solve the system of Bellman equations

$$Q^\pi(s, a) = \mathcal{R}(s, a) + \gamma \sum_{s}^{\prime} P(s, a, s') Q^\pi(s', \pi(s')) \qquad (57)$$

where Q^π, and R are constructed as vectors of size $|\mathcal{S}||\mathcal{A}|$ and P is a matrix of size $|\mathcal{S}||\mathcal{A}| \times |\mathcal{S}||\mathcal{A}|$ which describes the transition from (s, a) to s', a'. The transitions probabilities P, and expected rewards $\mathcal{R}(s, a) = \sum_s^{\prime} P(s, a, s') R(s, a, s')$

are unknown and hence are approximated from experience, and the approximated parameters w are found analytically.

LSPI then performs policy iteration through $\pi^{t+1}(s) = \text{argmax}_a Q(s,a)$, using the Q found in LSQ. This policy iteration is then performed until the performance improvement falls below some epsilon ($|w^{t+1} - w^t| < \epsilon$).



```
LSPI ($k$, $\phi$, $\gamma$, $\epsilon$, $\pi_0$, $D_0$)

    // $k$ : Number of basis functions
    // $\phi$ : Basis functions
    // $\gamma$ : Discount factor
    // $\epsilon$ : Stopping criterion
    // $\pi_0$ : Initial policy, given as $w_0$, $\pi_0 = \pi(s, w_0)$ (default: $w_0 = 0$)
    // $D_0$ : Initial set of samples, possibly empty

    $D = D_0$
    $\pi' = \pi_0$     // In essence, $w' = w_0$

    repeat
        Update $D$ (optional)               // Add/remove samples, or leave unchanged
        $\pi = \pi'$                          // $w = w'$
        $\pi'$ = LSQ ($D$, $k$, $\phi$, $\gamma$, $\pi$)          // $w'$ = LSQ ($D$, $k$, $\phi$, $\gamma$, $w$)
    until ($\pi \approx \pi'$)                     // that is, ($||w - w'|| < \epsilon$)

    return $\pi$                           // return $w$
```

Figure 5: LSPI algorithm taken from [13]

A new type of algorithm was introduced recently which rather than searching over value functions, searches directly over the policy space for an optimal policy. Ng and Jordan's algorithm PEGASUS [21] attempts to find a policy $\hat{\pi}$ which is close to

$$opt(M, \Pi) = \sup_{\pi \in \Pi} V_M(\pi). \tag{58}$$

Here, Π is the set of all policies, M is an MDP, and

$$V(\pi) := \mathbb{E}_{s_0 \sim D}(V^{\pi}(s_0)) \tag{59}$$

where D is a distribution of initial states. That is, the value $V(\pi)$ of a policy π is defined to be the expected state value over all start states $s_0 \sim D$. Obviously, however, policy search is impossible in MDP's with stochastic state transitions. The authors hence transform the stochastic MDP to a deterministic one, and perform policy search in the new MDP.

7.2 Gradient Based Methods

Many reinforcement learning algorithms, in particular the earlier ones, estimate a value function using gradient based methods. [29] introduce gradient descent for function approximation as a means of generalizing in large state action spaces. One drawback of these methods is that they use coding, which itself can be quite inefficient in large state and action spaces. [4] further highlight some issues with RL using function approximation in general. These issues pertain to the incremental acquisition of data (as opposed to learning from a training set), the sensitivity to error in bootstrapping methods (also discussed in [28]), the evolution of the estimation of Q^π, and the possibility of choosing a poor policy π even if the optimal Q^* is known. Despite these issues, however, practitioners continue to use gradient descent based methods [30] and numerous cases can be cited in which it has succeeded [4].

Recall the LSTD algorithm from the previous section. Sutton defines the TD(0) error as [30]

$$Err(R_t, s_t, a_t) = r + \gamma V(s') - V(s) = 0 \tag{60}$$

using the linear function approximator $Q(s,a) = \langle w, \phi(s) \rangle$. Note here the purposeful omission of the time index. This is because we assume the samples are i.i.d and hence the individual tuples $\phi(s), r, \phi(s')$ are considered. Now the solution to LSTD(0) [6], as well as the target value for TD(0) [29], is given by

$$err(r,s) = r + \gamma V(s') - V(s), \tag{61}$$

$$\mathbb{E}\big(err(r,s)\phi(s)\big) = d - Cw = 0 \tag{62}$$

where

$$d = \mathbb{E}\big(r\phi(s)\big) \tag{63}$$

$$C = \mathbb{E}\big(\phi(s)\left[\phi(s) - \gamma\phi(s')\right]^T\big). \tag{64}$$

Now if w does not satisfy equation 62 then TD learning may diverge [30]. Sutton considers $\mathbb{E}\big(err(r,s)\phi(s)\big)$ to be the error of the TD solution for w, and hence proposes a gradient descent algorithm which minimizes it. Firstly we recognize, as previously stated, that the vector $\mathbb{E}\big(err(r,s)\phi(s)\big)$ must be zero for a TD solution, and hence that its vector norm $\left[\mathbb{E}\big(err(r,s)\phi(s)\big)\right]^T \left[\mathbb{E}\big(err(r,s)\phi(s)\big)\right]$ represents the current distance from the solution. With this target, the objective function is formed:

$$w^* = \underset{w}{\operatorname{argmin}}\, J(w) \tag{65}$$

$$= \underset{w}{\operatorname{argmin}} \left[\mathbb{E}\big(err(r,s)\phi(s)\big)\right]^T \left[\mathbb{E}\big(err(r,s)\phi(s)\big)\right]. \tag{66}$$

Now this can be solved using gradient descent where

$$\nabla_w J(w) = -2\mathbb{E}\left[\phi(s)\big(\phi(s) - \gamma\phi(s')\big)^T\right]^T \mathbb{E}\big(err(r,s)\phi(s)\big) \tag{67}$$

Now a simple gradient descent could be performed if it weren't for the caveat that the two expectations cannot be sampled together since their correlation will bias the outcome. Instead, one could store a long term estimate of one and sample the other. It is shown to be possible to store the long term estimate of the first expectation, however it is also shown to take $O(n^2)$ computation at each time step. Although this is not horrendously expensive, the opposite (storing the second expectation) only takes $O(n)$ computation per time step. Hence, we store the long run estimate of $\mathbb{E}\big(err(r,s)\phi(s)\big)$ as

$$u_{k+1} = u_k + \beta_k\big(err(r_k, s_k)\phi(s_k) - u_k\big) \tag{68}$$

for some forgetting weight $\beta \in (0,1]$ which weights toward more recent values for higher β. The gradient descent step is then performed through

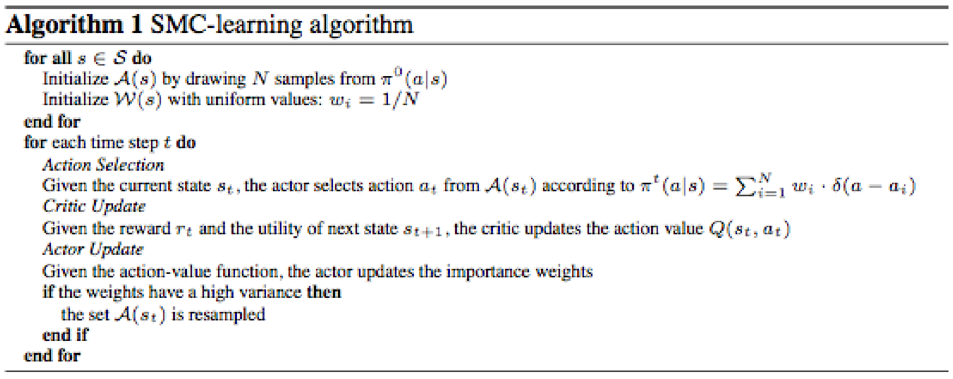
$$w_k + 1 = w_k + \alpha_k\big(\phi(s_k) - \gamma\phi(s_k')\big)\phi(s_k)^T u_k \tag{69}$$

8 Approximating The Problem (MDP)

Some (few modern) RL methods have been proposed which, through some discretization of the state(/action) space, approximate the underlying MDP. That is, by discretizing, the state space is transformed from a continuous MDP, to a discrete one in which the state transition function, and reward function can now be approximated. Convergence to the optimal value function can now be guaranteed in the new MDP, however there is no guarantee of how the optimal policy in the new MDP relates to that in the original. Hence, it is generally considered better to approximate the solution (V in the original MDP) than to approximate the problem.

A sequential Monte-Carlo algorithm was introduced to deal with continuous action spaces in [16]. This actor-critic method stores a stochastic policy $\pi(a|s)$ which is a discrete probability distribution from which a possible action set $\mathcal{A}(s) = \{a_1, a_2, \ldots, a_N\}$ is drawn, at a state s. This is a novel way of discretizing the action space. Through doing this, however, the algorithm transforms the original MDP into a new one in which only the actions $a \in \mathcal{A}(s)$ may be chosen. Hence, an optimal solution can be found in the new MDP, but with no guarantees about its optimality in the original MDP. Practically speaking, this means that the algorithm is susceptible to missing important parts of the action space.

In this algorithm, actions a_i with high importance weights w_i are chosen more frequently. The action to be taken at each step is then chosen at random from this set $\mathcal{A}(s)$. Hence actions with higher weights are chosen more frequently than those with lower weights. Further, the initial importance weights are all set equal such that actions are chosen uniformly, promoting exploration in the early stages of learning.



Algorithm 1 SMC-learning algorithm

```
for all $s \in \mathcal{S}$ do
    Initialize $\mathcal{A}(s)$ by drawing $N$ samples from $\pi^0(a|s)$
    Initialize $\mathcal{W}(s)$ with uniform values: $w_i = 1/N$
end for
for each time step $t$ do
    Action Selection
    Given the current state $s_t$, the actor selects action $a_t$ from $\mathcal{A}(s_t)$ according to $\pi^t(a|s) = \sum_{i=1}^N w_i \cdot \delta(a - a_i)$
    Critic Update
    Given the reward $r_t$ and the utility of next state $s_{t+1}$, the critic updates the action value $Q(s_t, a_t)$
    Actor Update
    Given the action-value function, the actor updates the importance weights
    if the weights have a high variance then
        the set $\mathcal{A}(s_t)$ is resampled
    end if
end for
```

Figure 6: Sequential Monte-Carlo algorithm taken from [16]

9 Dealing Directly With Continuous Spaces

In this section I contrast those algorithms which discretize continuous spaces with those which don't. It is important to note here the difference between discretizing and approximating the problem. Many early algorithms discretized the state space and found the exact solution in the discretized space. In this section the discretized algorithms I describe assume a discretized value function, however the agent is still acting in the original continuous MDP. Hence these algorithms find an approximate solution in the original MDP.

9.1 Non-Discretized Algorithms

Much of the reinforcement learning literature to date [29] has dealt with finite MDPs, with successful application to small toy problems [27] – that is they have only worked in discrete action/state spaces (as claimed by [16]).

Although the function approximation methods introduced in [29] are introduced with such techniques as coding or tiling as a means of discretization, there is no obvious argument against directly performing the methods with feature maps calculated directly from the continuous states and actions. Of methods both old and new, however, there are few which have been proposed to directly manage continuous state and action spaces.

LSTD and LSPI have been proposed to fit the data, without using gradient based methods, for discrete state and action spaces. More recently, an interesting algorithm for continuous multi-dimensional action spaces was introduced in [1]. This algorithm, named fitted q iteration (FQI), finds the state-action value function, Q_{k+1}, which best fits the regression problem between the list of pairs:

$$\left\{ \left[(s_t, a_t), r_t + \gamma \max_{a^prime \in \mathcal{A}} Q_k(s_{t+1}, a') \right]_{1 \leq t \leq N} \right\}. \tag{70}$$

This list of pairs is found through experience, following a fixed *behavior* policy distribution $\pi_b(\cdot|s)$.

One example of FQI uses least squares regression, and as such has been named least squares fitted q iteration [1]. The Q-iteration is then done through

$$Q_{k+1} = \underset{Q \in \mathcal{F}}{\operatorname{argmin}} \sum_{t=1}^{N} \frac{1}{\pi_b(a_t|s_t)} \left(Q(s_t, a_t) - \left[r_t + \gamma \max_{a' \in \mathcal{A}} Q_k(s_{t+1}, a') \right] \right)^2, \tag{71}$$

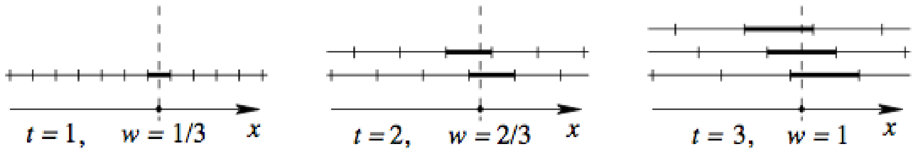
where the normalizer term $\frac{1}{\pi_b(a_t|s_t)}$ ensures that there is no bias toward actions which are favorable in the behavior policy. Further, $\mathcal{F}$ is a set of functions from which $Q(s_t, a_t)$ can be approximated. For example $\mathcal{F}$ may be the set of functions defined by a linear function approximator.

9.2 Discretized Algorithms

Traditionally reinforcement learning algorithms cannot act directly in continuous states, actions and time. In this section we discuss the algorithms which discretize continuous spaces, and the ways in which they do so.

An in depth study of tile coding is given in [27]. Tile coding [27, 29] represents the state using binary features, which when switched on show in which tiles the state lies, as shown in Figure 7. Although this study is thorough, the experimental section was not convincing as it worked entirely with a discrete gridworld problem. The only continuous contribution was a real valued dimension added to the action which determined the probability of the selected action being taken.

Alternatively to binary tile coding, are the continuous real valued features described by radial basis functions. This style of "discretization" sets up a



$t = 1,\quad w = 1/3 \quad x$
$t = 2,\quad w = 2/3 \quad x$
$t = 3,\quad w = 1 \quad x$

Figure 7: Example of a 1-dimensional tile coding taken from [27]

network of n gaussians θ on the state space, and represents the state as an n-dimensional vector $\phi(s)$, in which each dimension i describes the proximity of s to θ_i. A novel type of algorithm was presented by [4] (discussed earlier), in which the RBFs are adjusted according to the input data and the error. Further, in this algorithm, RBFs can be added if insufficient information is known about the current state. Figure 8 gives an example of a radial basis function with possible adjustments which can be made according to the error.

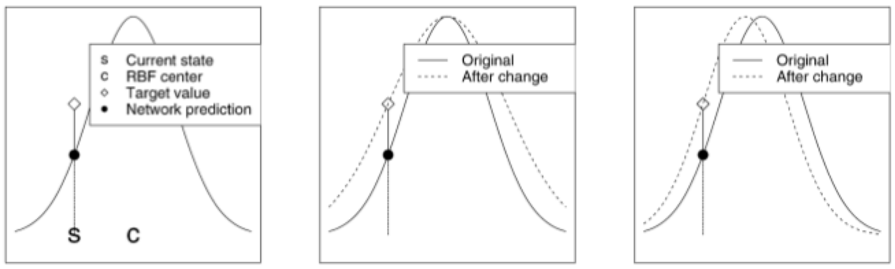


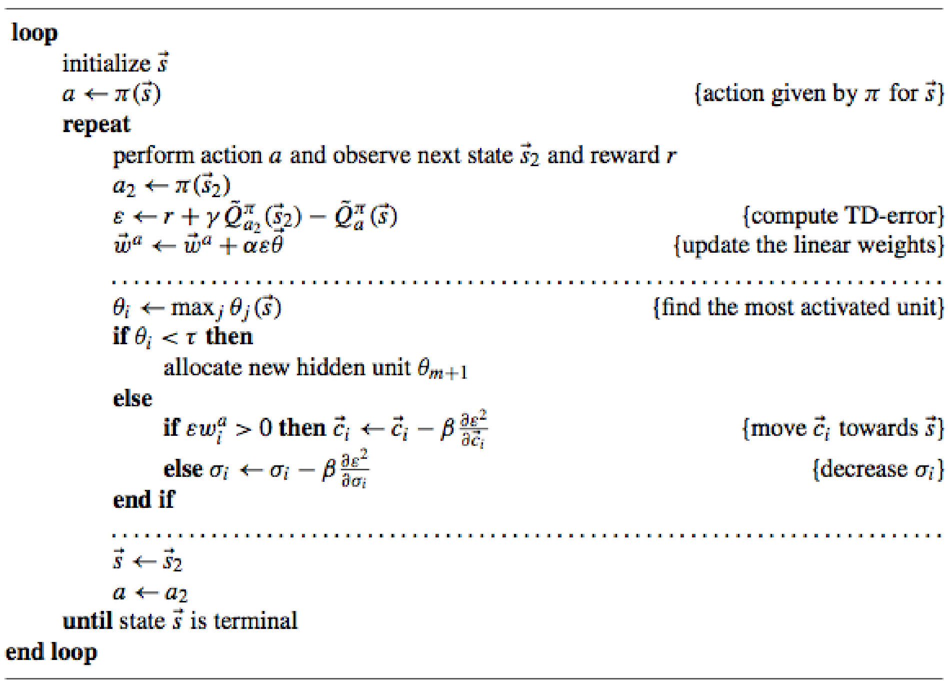
Figure 8: Example of a RBF, and example changes taken from [4]

Barreto and Anderson [4] presented a Restricted Gradient Descent (RGD) which represents the states using radial basis functions. That is, each dimension of the feature map $\phi(s)$ is a radial basis function $\theta_i(s)$, with width σ_i. The parameters w, to function $Q(s, a) = \sum_i w_i^a \theta_i(s)$, are updated using stochastic gradient descent on the TD error as presented previously. The algorithm then performs a novel step by adjusting the RBFs according to the error. More specifically, if the state is within a specified proximity τ of the nearest RBF, then the centre and width of the RBF is adjusted. Alternatively, if $\theta(s) > \tau$ an RBF is added at s. Adjusting the RBFs in this way makes their size reflect the value function [4].

9.3 Exploring In Continuous State Spaces

In 2008 multi-resolution exploration was introduced as a means for exploring a continuous state space [22]. In this algorithm the continuous state space is descretized in a unique, multi-resolutional way. As various regions of the state space become more frequently explored the resolution of the discretization is made finer, and hence the estimate of the value function becomes more and more accurate in better explored areas.

They provide explanation of how this idea can be applied to LSPI, FQI, and



```
loop
    initialize $\vec{s}$
    $a \leftarrow \pi(\vec{s})$                                   {action given by $\pi$ for $\vec{s}$}
    repeat
        perform action $a$ and observe next state $\vec{s}_2$ and reward $r$
        $a_2 \leftarrow \pi(\vec{s}_2)$
        $\varepsilon \leftarrow r + \gamma \tilde{Q}^{\pi}_{a_2}(\vec{s}_2) - \tilde{Q}^{\pi}_{a}(\vec{s})$          {compute TD-error}
        $\vec{w}^a \leftarrow \vec{w}^a + \alpha\varepsilon\vec{\theta}$                   {update the linear weights}
        ..........................................................................
        $\theta_i \leftarrow \max_j \theta_j(\vec{s})$                          {find the most activated unit}
        if $\theta_i < \tau$ then
            allocate new hidden unit $\theta_{m+1}$
        else
            if $\varepsilon w^a_i > 0$ then $\vec{c}_i \leftarrow \vec{c}_i - \beta\frac{\partial\varepsilon^2}{\partial\vec{c}_i}$             {move $\vec{c}_i$ towards $\vec{s}$}
            else $\sigma_i \leftarrow \sigma_i - \beta\frac{\partial\varepsilon^2}{\partial\sigma_i}$                         {decrease $\sigma_i$}
        end if
        ..........................................................................
        $\vec{s} \leftarrow \vec{s}_2$
        $a \leftarrow a_2$
    until state $\vec{s}$ is terminal
end loop
```

Figure 9: RGD algorithm taken from [4]

a model based approach. Further empirical results are given when applying the model based approach to the mountain car problem.

Part III

Application Domains

In robotics, it is obvious how continuous observations are received through various sensors. Continuous actions are also important. Consider for example the juggling robot [26] which has a six dimensional state input with various joint angles and torques, and has a multidimensional continuous action which manipulates these quantities.

[15] proposed that interactive game AI is the perfect application for human level AI. They describe human level AI as seamlessly integrating "all the human capabilities", citing examples such as HAL from "2001, a Space Odyssey", and C-3PO and R2D2 from "Star Wars". In this work I do not endeavour to create human level AI, however, I do take the thesis that interactive game AI is a good testbed for real world reinforcement learning techniques.

[9] successfully applied reinforcement learning to a fight game – Tao Feng – citing implementation as the most difficult part due to the required understanding of the game engine. They used SARSA learning, as described above, with function approximation. That is, the error in the function approximator was given by:

$$\begin{aligned} Err(s,a;t) &= R_t - Q(s_t,a_t) && (72)\\ &= r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t). && (73) \end{aligned}$$

This success sparked interest in reinforcement learning for games, with papers emerging about RL in first person shooters [19, 18, 14], SecondLife [20], and Super Mario Bros [33].

In a custom built first person shooter, [19, 18] obtained reasonable results using SARSA(λ), with tabular value representation. These bots obtained higher accuracy than rule based methods, through firing less shots but making them more accurate. Hence the RL bots scored fewer kills, and further suffered more deaths.

[20] points out that massively multiplayer online role playing games (MMORPG) are have a much stronger need for non-player characters. This paper states that MMORPGs are played over months, rather than hours like other computer games, hence creating the need for non-player support characters. It further points out that simple rule-based non-player characters have stationary AI, and do not evolve and learn with a constantly changing environment.

Non-player characters in MMORPGs are categorized by [20] into reflexive agents and learning agents. Reflexive agents are those which simply react to the current state by reflex, with a built in set of rules. Learning agents on the other hand "are able to modify their internal structure in order to improve their performance with respect to some task".

Using a motivation function with Q-learning, the agents in the MMORPGs are able to adapt their skills by a constantly changing reward function. This means that unlike the fighting agent in Tao Feng, which are restricted to learning to fight due to their static reward function, this agent can learn new skills based on the current reward function.

Further, [33] studied the application of reinforcement learning to the game Super Mario Brothers, giving similar justification to [15] for choosing the interactive game domain – “we want to see what our learning algorithms and function representations are capable of” [33]. The author further points out that different games require different sets of skills, with different gameplay, and different learning techniques. Hence, he claims, game AI is the perfect testbed for testing the versatility of our RL algorithms.

Further, we could extend continuous state/action RL to planning and scheduling tasks, which have previously been done in continuous state spaces [24], but not continuous multidimensional action spaces.

Part IV
Where To From Here?

10 Motivation and Contributions

Real world reinforcement learning problems are continuous (and typically multidimensional) in both state and action spaces. Consider for example robotics, interactive game AI, and planning and scheduling, all of which can observe a continuous state signal, possibly perform a continuous multidimensional action, and perhaps even exist in continuous time. Traditional RL, on the contrary, has a main focus on finite MDPs with issues arising even when trying to scale to large finite MDPs. Further, very little work has been directed towards working directly in continuous spaces, much less dealing with issues scaling to *large* continuous MDPs. We identify the following three areas of research in continuous RL

- Traditionally, function approximation has been used for generalizability in large spaces [29]. A simple continuous feature representation is compact, yet scarcely explored in the literature. The questions then arise of how to choose appropriate features, how to apply kernels, and which loss function to use (traditionally only the squared loss has been used loss).

- Further problems arise when dealing with multidimensional continuous action spaces. This becomes difficult as we need to be able to find expressive Q-functions and loss functions for which we can optimize both the parameters, and the actions. That is, we are required to optimize parameters in policy evaluation, and optimize actions in policy iteration. Hence, the loss function must be able to optimize each. One possible (restrictive) solution is to enforce our loss to be convex in both parameters and actions.

- One possible area to start is generalizing the least squares temporal difference (LSTD) and least squares policy iteration (LSPI) algorithms from the discrete case to continuous. These algorithms are well motivated as they calculate the parameters analytically, alleviating need for fine tuned gradient descents. These algorithms, however have not been written to deal with continuous spaces.

I intend to focus my thesis on reinforcement learning in continuous state and action spaces, dealing with the above mentioned scalability issues. To do this I need to address the following problems (and more):

- How to construct features,

- How to construct loss function,

- How to construct Q function,

- How to effectively explore, allowing agent to experience success.

Further, a contribution can be made by generalizing the LSTD and LSPI algorithms to continuous spaces. Once these contributions have been made, I wish to explore the problems associated with scalability in large problems. It is essential that this is solved before RL can be applied to real world problems.

I will begin by applying my algorithms to the mountain car and cart pole problems, and when exploring the scalability issues, possibly extend to the tennis interface.

11 Proposed Progress

I commenced my PhD. in November 2008, and hence must be close to completion by November 2011. With this time constraint, I propose the below timeline:

- NOV 2009: Developed tennis interface, mountain car, and cart pole simulaters. Begun work on algorithms gaining good understanding of the underlying problems.
- DEC 2009: Have working algorithm for intelligent agents in mountain car and cart pole using polynomial kernels.
- Jul 2010: Submit paper on algorithm.
- JAN-FEB 2010: Begin work modifying LSPI and LSTD to continuous domains.
- JUN 2010: Further explore the RGD algorithm, and begin extending it. Be on the way to a paper on extended LSPI and LSTD.
- Further revise from here.

A Notation

In this text I adopt a notation very similar to that in [29]:

notation	description
t	a discrete time
T	t at termination of episode
s_t	agents state at time t
a_t	agents action at time t
$r_t = R(s_{t-1}, a_{t-1}, s_t)$	reward received by agent at time t as a result of taking action a_{t-1} in state s_{t-1} and ending in state s_t
R_t	Return at time t generally given by some model of optimality. eg. $R_t = \sum_{i=t}^{T} \gamma^{t-1} r_t$
π^*	Optimal policy.
$V(s)$	The "value", or expected return, in state s
$Q(s, a)$	The "value", or expected return, of following action a in state s
$V^\pi(s)$	The "value", or expected return, in state s when following policy π
$Q^\pi(s, a)$	The "value", or expected return, of following action a in state s and policy π thereafter.
$V^*(s)$	The "value", or expected return, in state s under the optimal policy π^*
$Q^*(s)$	The "value", or expected return, of following action a in state s and following optimal policy π^* thereafter.

References

[1] A. Antos, R. Munos, and C. Szepesvári. Fitted q-iteration in continuous action-space mdps. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *NIPS*. MIT Press, 2007.

[2] P. Auer, N. Cesa-Bianchi, P. Fischer, and L. Informatik. Finite-time analysis of the multi-armed bandit problem, 2000.

[3] L. Baird and A. Moore. Gradient descent for general reinforcement learning. In *In Advances in Neural Information Processing Systems 11*, pages 968–974. MIT Press, 1998.

[4] A. Barreto and C. Anderson. Restricted gradient-descent algorithm for value-function approximation in reinforcement learning. *Artificial Intelligence*, 172:458–482, 2008.

[5] B. Bouzy and G. Chaslot. Monte-carlo go reinforcement learning experiments. *IEEE 2006 Symposium on Computational Intelligence in Games*, pages 187–194, 2006.

[6] J. Boyan. Least-squares temporal difference learning. In *In Proceedings of the Sixteenth International Conference on Machine Learning*, pages 49–56. Morgan Kaufmann, 1999.

[7] E. Even-Dar and Y. Mansour. Learning rates for q-learning. In *Fourteenth Annual Conference on Computational Learning Theory (COLT)*, 2001.

[8] I. Ghory. Reinforcement learning in board games, 2004.

[9] T. Graepel, R. Herbrich, and J. Gold. Learning to fight. In *In Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, 2004.

[10] M. Harmon and S. Harmon. Reinforcement learning: a tutorial. http://www.nbu.bg/cogs/events/2000/Readings/Petrov/rltutorial.pdf, 1996.

[11] L. Kaelbling. *Learning in embedded systems.* The MIT Press, 1993.

[12] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[13] M. G. Lagoudakis and R. Parr. Model-free least squares policy iteration. Technical report, Advances in Neural Information Processing Systems, 2001.

[14] J. Laird and J. Duchi. Creating human-like synthetic characters with multiple skill levels: A case study using the soar quakebot. American Association for Artificial Intelligence, 2000.

[15] J. E. Laird and M. van Lent. Human-level ai's killer application: Interactive computer games. Association for the Advancement of Artificial Intelligence, 2000.

[16] A. Lazaric, M. Restelli, and A. Bonarini. Reinforcement learning in continuous action spaces through sequential monte carlo methods. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *NIPS*. MIT Press, 2007.

[17] H. Mannen. Learning to play chess using chess with database games, 2003.

[18] M. McPartland and M. Gallagher. Creating a multi-purpose first person shooter bot with reinforcement learning. In *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games*, 2008.

[19] M. McPartland and M. Gallagher. Learning to be a bot: Reinforcement learning in shooter games. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, October 22-24, 2008, Stanford, California, USA*, 2008.

[20] K. Merrick and M. L. Maher. Motivated reinforcement learning for non-player characters in persistent computer game worlds. In *ACE '06: Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology*, page 3, New York, NY, USA, 2006. ACM.

[21] A. Ng and M. Jordan. Pegasus: A policy search method for large mdps and pomdps. In *In Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pages 406–415, 2000.

[22] A. Nouri and M. L. Littman. Multi-resolution exploration in continuous spaces. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *NIPS*, pages 1209–1216. MIT Press, 2008.

[23] R. E. Parr. Hierarchical control and learning for markov decision processes, 1998.

[24] J. Rintanen, B. Nebel, J. C. Beck, and E. A. Hansen, editors. *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia, September 14-18, 2008.* AAAI, 2008.

[25] H. Robbins. Some aspects of sequential design of experiments. *Bulletin of the American Mathematical Society*, 58:527–535, 1952.

[26] S. Schaal and C. Atkeson. Robot juggling: implementation of memory-based learning. *Control Systems Magazine, IEEE*, 14(1):57–71, 1994.

[27] A. A. Sherstov and P. Stone. Function approximation via tile coding: Automating parameter choice. In J.-D. Zucker and I. Saitta, editors, *SARA 2005*, volume 3607 of *Lecture Notes in Artificial Intelligence*, pages 194–205. Springer Verlag, Berlin, 2005.

[28] W. D. Smart and L. P. Kaelbling. Practical reinforcement learning in continuous spaces. pages 903–910. Morgan Kaufmann, 2000.

[29] R. S. Sutton and A. G. Barto. *Reinforcement Learning.* The MIT Press, 2001.

[30] R. S. Sutton, C. Szepesvári, and H. R. Maei. A convergent o(n) temporal-difference algorithm for off-policy learning with linear function approximation. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *NIPS*, pages 1609–1616. MIT Press, 2008.

[31] I. Szita and A. Lőrincz. The many faces of optimism: A unifying approach. In *International Conference on Machine Learning*, 2008.

[32] G. Tesauro. Practical issues in temporal difference learning. In *Machine Learning*, pages 257–277, 1992.

[33] J. Togelius, S. Karakovskiy, J. Koutnik, and J. Schmidhuber. Super mario evolution. In *2009 IEEE Transactions on Computation Intelligence and Games*, 2009.

[34] M. van Otterlo. Markov decision processes: Concepts and algorithms, 2008.

[35] C. Watkins. Learning from delayed rewards, 1989.