8-2014

Convergence of a Reinforcement Learning Algorithm in Continuous Domains

Stephen Carden
Clemson University, swcarde@clemson.edu

Convergence of a Reinforcement Learning Algorithm in Continuous Domains

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Mathematical Sciences

---

by
Stephen Wesley Carden
August 2014

Accepted by:
Dr. Peter Kiesler, Committee Chair
Dr. Robert Lund
Dr. Colin Gallagher
Dr. Brian Fralix

Abstract

In the field of Reinforcement Learning, Markov Decision Processes with a finite number of states and actions have been well studied, and there exist algorithms capable of producing a sequence of policies which converge to an optimal policy with probability one. Convergence guarantees for problems with continuous states also exist. Until recently, no online algorithm for continuous states and continuous actions has been proven to produce optimal policies.

This dissertation contains the results of research into reinforcement learning algorithms for problems in which both the state and action spaces are continuous. The problems to be solved are introduced formally as Markov Decision Processes. Also introduced is a value-function solution method known as Q-learning.

The primary result of this dissertation is the presentation of a Q-learning type algorithm adapted for continuous states and actions, and the proof that it asymptotically learns an optimal policy with probability one. While the algorithm is intended to advance the theory of continuous domain reinforcement learning, an example is given to show that with appropriate exploration policies, it can produce satisfactory solutions to non-trivial benchmark problems.

Kernel regression based algorithms have excellent theoretical properties, but have high computational cost and do not adapt well to high-dimensional problems. A class of batch-mode regression tree-based algorithms is introduced. These algorithms are modular in the sense that different methods for partitioning, performing local regression, and choosing representative actions can be chosen. Experiments demonstrate superior performance over kernel methods.

Batch algorithms possess superior computational efficiency, but pay the price of not being able to use past observations to inform exploration. A data structure useful for limited learning during the exploration phase is introduced. It is then demonstrated that this limited learning can outperform batch algorithms using totally random action exploration.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Consider a person searching for a lost puppy in an unfamiliar heavily wooded area. Due to the thickness of the brush, there is a limited number of paths the person may travel. The cries of the puppy constitute a signal by which the person can judge their progress to the goal. The searcher's task reduces to choosing paths to try and minimize the time until the puppy is found.

By considering elements of the above scenario in the abstract, one can create a class of problems and consider methods for learning good decisions. The four key structures are the state space (describing the environment of the agent), the action space (describing the decisions the agent must choose from), the reward signal (numeric feedback about the most recent state and action), and the state transition structure (governing how states and actions lead to new states).

Reinforcement Learning is defined broadly by Sutton & Barto [29] as "learning what to do—how to map situations to actions—so as to maximize a numerical reward signal... Moreover, it is usually assumed from the beginning that the agent has to operate despite significant uncertainty about the environment it faces." The latter half of their statement reveals the role that statistics can play in Reinforcement Learning. The agent almost never has access to information concerning the dynamics of the system, so a priori planning is not a realistic option. Instead, the agent must use trial and error, collect data about how the system operates, and use that data intelligently towards good decisions.

Let us consider four difficulties associated with learning to find the puppy in minimum time. The ultimate reward, finding the puppy, can be far delayed from crucial path selections. If an initial search is unsuccessful, the searcher must be able to assign value to early decisions distant in time

from the final goal. Also, the searcher may find that path selection based only on audio cues may lead to dead ends or repeated loops. It may be advantageous to sometimes choose paths for the sake of exploration—to try and find paths better than any currently known. Third, due to wind, echoes, or other complications, audio signals from the puppy may be influenced by random factors. The searcher must be able to "average" out the noise and estimate the true signal. Finally, if the area to be searched is large and there are many paths, the amount of memory required may overwhelm the searcher.

The four difficulties cited in the puppy-search problem are central to Reinforcement Learning. First, the magnitude of rewards may depend greatly on actions used many time steps before the reward is received, so there must be some method for relating states and actions to delayed rewards. To highlight the importance of this problem, consider that the dissertation in which Q-learning, a staple of modern Reinforcement Learning, was introduced was titled "Learning from Delayed Rewards" [36]. Later we will see how ideas from Dynamic Programming can assist with this problem.

The second chief difficulty is achieving a balance between exploration and exploitation. On one hand, if the agent has, in the past, experienced high rewards when using a particular action in a particular state, it seems reasonable to select that action again upon return to that state. However, it may be that there are other actions which have not yet been tried that yield higher reward, that other attempted actions have higher average reward but random observations thus far have not revealed that, or that actions with low immediate reward can lead to future states with the potential of higher reward. For this reason, it is necessary to sometimes choose actions which appear to be sub-optimal in order to sufficiently explore state and action possibilities.

The third difficulty is that the system may be governed by stochastic dynamics. Trying an action twice from the same state may not produce identical rewards or transitions. This is where statistics enters the problem. Rewards must be processed as expectations, and transitions must be processed as probabilities. Furthermore, as values of states and actions will depend on the values of subsequent states, estimates of expectations and probabilities must remain fluid and adjust as estimates for other states and actions change.

The fourth difficulty will receive the most attention in this dissertation. If the number of states and actions is small, the Reinforcement Learning problem is essentially solved both in theory and practice. However, most interesting real-life problems involve many variables, with a large or

even uncountable number of state and action combinations. In such situations, estimating the value of each state and action is infeasible, and some form of knowledge generalization is necessary. Here again, statistics has an opportunity to contribute because the value estimate generalization problem is, at its core, a regression problem.

The remainder of this introduction will formalize the concepts relating to Reinforcement Learning problems, as well as introduce a classical solution method. Chapter 2 will examine theoretical properties of a new algorithm modified for continuous state and action variables. Chapter 3 takes a more pragmatic approach and introduces computationally efficient data structures for value estimate regression. Chapter 4 introduces a new exploration strategy that is particularly helpful for large problems which are difficult to explore randomly.

1.1 Discrete Markov Decision Processes

A Markov Decision Process (MDP) is a stochastic control problem, usually considered in discrete time. At each time step, the system is in a state, and a controller chooses an action. The system then transitions to a new state with a probability distribution that depends both on the previous state and the action utilized. A reward (or cost, for some authors) is also received. The rewards are allowed to be random variables with a distribution that depends on the state and the action utilized. The goal is to find a policy (a function that maps states to actions) that maximizes rewards (or minimizes costs) according to some measure of goodness. We will consider MDPs for which there are no terminal, absorbing states.

This section will formally introduce finite Markov Decision Processes. There are four components:

- S, the state space, describing the states or situations the system may find itself in. For now, we assume S is finite.
- A, the action space, describing the actions or decisions the controller may choose from. For now, we assume A is finite.
- $P(s, s', a)$, a set of transitions giving the probability of transitioning to state s' when the system is in state s and the controller chooses action a.

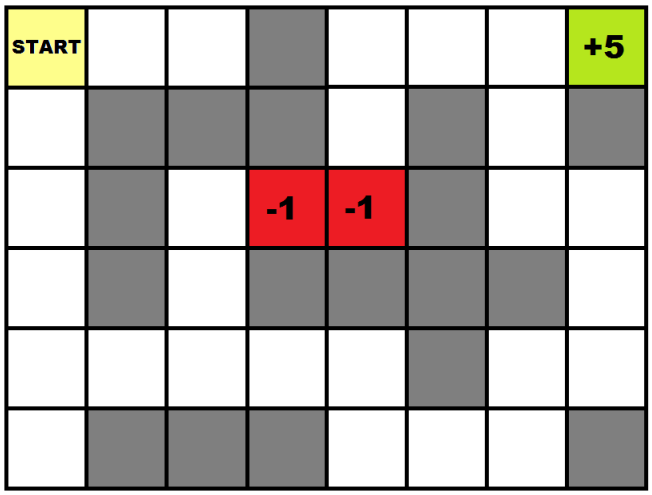

Figure 1.1: A simple deterministic MDP. The agent must learn to reach the goal state while avoiding the states with high penalties.

- $R(s, a)$, a set of random variables describing the reward received when the system is in state s and the controller chooses action a. Rewards are bounded by a constant C_0.

The Markov Decision Process proceeds as follows: the process begins in some state $s \in S$ and the controller chooses an action $a \in A$. A random reward $R(s, a)$ is received and the system transitions to a new state in accordance with the probability measure $P(s, \cdot, a)$. The system progresses to the next time step, and this scenario repeats from the new state. The reward distributions and transition probabilities are assumed to have the Markov property: they only depend on the current state and action; they do not depend on the history of previous states and actions.

For full generality, we allow mechanics to be stochastic, but they may be deterministic. Figure 1.1 gives an example of a simple deterministic MDP in which an agent must traverse a maze to find a goal state which emits rewards. There are two possible paths the agent may traverse, but one contains penalty states with negative rewards. To behave optimally, the agent must learn to take the path without penalties.

Figure 1.2 represents the "Left-Right Control" problem, a simple process with stochastic mechanics. The agent starts in state x_0 and must try to steer the system towards one of two reward states, x_1 or x_5. The actions available, "left" and "right," tend to steer the system in the direction of their name, but with a small probability will move in the opposite direction. It is obvious that the agent should usually use action "right" and try to obtain the higher reward at state x_5, but

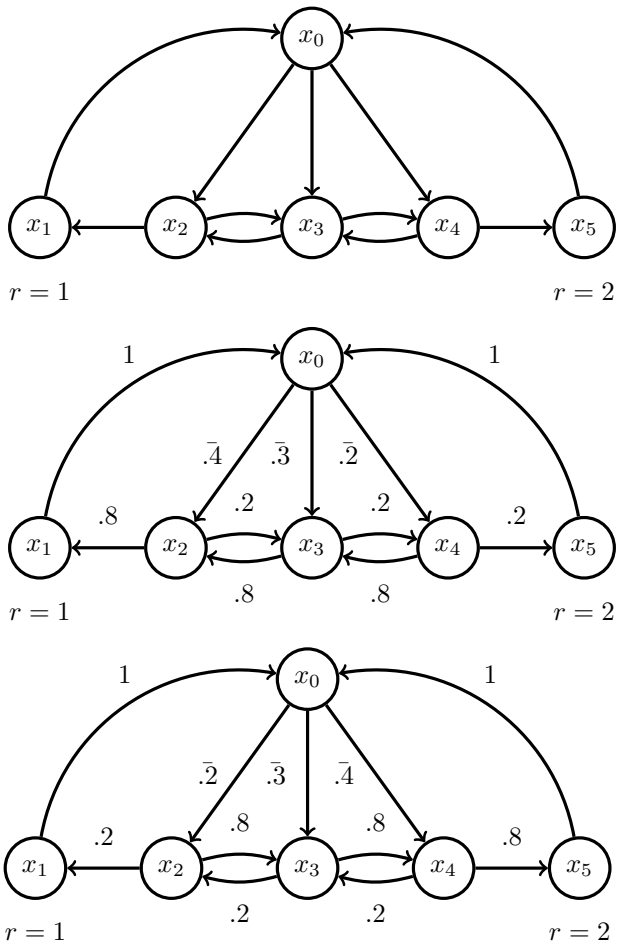

Figure 1.2: A simple MDP with stochastic dynamics. The first diagram shows the states and their associated rewards. The second diagram shows transition probabilities under action "left," and the last diagram shows transition probabilities under action "right."

suppose the system transitions to state x_2. Is it now better to use action "left" and receive a reward of 1 in as few as 1 time step, or is it more desirable to use action "right" and try to obtain a reward of 2 in a minimum of 3 time steps? This highlights the need to be specific about how goodness is measured as a function of rewards and when they are received.

To that end, define a decision-making policy π as a function $\pi : S \to A$. In some settings, policies are allowed to be stochastic or non-stationary, but we will restrict our attention to deterministic policies. Define $P_\pi(s, s') := P(s, s', \pi(s))$. Notice P_π is a transition kernel for a Markov chain on the space S.

Let $\gamma, 0 < \gamma < 1$, be a discount factor. γ represents the loss in value for delayed rewards. A value of γ close to one means that rewards received in the future retain almost all their value, while a lower value for γ means the value of a reward decays quickly over time. Let (s_t, a_t) be the (random) state and action at time t. Under a policy π, the value of a state s is defined under the expected total discounted reward criterion:

$$V^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t)) | s_0 = s\right]. \tag{1.1}$$

A policy π^* is optimal if it satisfies $V^{\pi^*}(s) = \sup_\pi V^\pi(s), \forall s \in S$.

If the transition probabilities and reward means are known, then dynamic programming methods can obtain an optimal policy in a finite number of steps [22]. However, if probabilities and rewards are initially unknown but transitions and rewards can be simulated or observed, then the problem becomes much more difficult, one amenable to the methods of Reinforcement Learning.

1.2 Q-learning

The goal is to find an algorithm that will, when certain assumptions are met, converge to an optimal policy. To that end, we consider *Q-values* [36] as

$$Q^\pi(s, a) := E[R(s, a)] + \gamma \sum_{t \in S} V^\pi(t) P(s, t, a). \tag{1.2}$$

Intuitively, $Q^\pi(s, a)$ is the value obtained if we start in state s, utilize action a, and then follow policy π thereafter. Consider the Q-values associated with an optimal policy π^*: $Q^*(s, a) := Q^{\pi^*}(s, a)$.

If we can learn the values $Q^*(s,a)$ for all (s,a), then an optimal policy can easily be recovered by setting

$$\pi^*(s) = \operatorname*{argsup}_{a \in A} Q^*(s,a).$$

Q-learning is an algorithm for producing values $\widehat{Q}_n(s,a)$ which estimate and converge to $Q^*(s,a)$ as $n \to \infty$. The algorithm requires a real-valued sequence of *learning parameters* α_k for each state-action pair such that

$$0 < \alpha_k < 1, \qquad \sum_{k=1}^{\infty} \alpha_k = \infty, \qquad \sum_{k=1}^{\infty} \alpha_k^2 < \infty. \tag{1.3}$$

The basic Q-learning algorithm proceeds as follows:

1. Initialize $\widehat{Q}_0(s,a) = 0 \ \forall (s,a) \in S \times A$.
2. Pick an initial state s_1.
3. For $n \geq 1$, repeat the following:
   (a) Select an action a_n, either by $a_n = \arg\max_{a' \in A} \widehat{Q}_{n-1}(s_n, a')$ or according to a predetermined exploration method.
   (b) Perform action a_n. Receive reward r_n and observe the state transitioned to, u_n.
   (c) Update Q-value estimates. Letting integer-valued $m(s,a)$ denote the number of visits to state (s,a), define

$$\widehat{Q}_n(s_n, a_n) := (1 - \alpha_{m(s_n,a_n)})\widehat{Q}_{n-1}(s_n, a_n) + \alpha_{m(s_n,a_n)} \left[r_n + \max_{a' \in A} \widehat{Q}_{n-1}(u_n, a') \right].$$

   (d) Set $s_{n+1} = u_n$.

Watkins proved the following theorem.

Theorem 1. *Given bounded rewards $|R(s,a)| < C_0 < \infty$, learning parameters that satisfy (1.3), and assuming all state-actions are visited infinitely often, then with probability one,*

$$\widehat{Q}_n(s,a) \to Q^*(s,a) \ \forall (s,a) \in S \times A$$

as $n \to \infty$.

This result is the foundation for what follows in subsequent chapters.

1.3 Exploration

Consider the instructions "Select an action a_n, either by $a_n = \arg\max_{a' \in A} \widehat{Q}_{n-1}(s_n, a')$ or according to a pre-determined exploration method." The particular method used to select actions is the primary influence on what parts of the state-action space are observed. On one hand, it seems advantageous to choose actions that have been observed to yield the highest known rewards, so that high-reward areas are explored fully. On the other hand, the highest known rewards may belong to a local, not global, maximum. Thus unless actions which appear initially to be sub-optimal are tried, the algorithm can be trapped in a subset of the state-action space which will not produce an optimal policy.

The method used to select actions is called the *exploration policy*. Here are a few common exploration policies:

- **Random** Choose actions randomly and uniformly from all possible actions. This method is suitable for small problems, but for larger problems it can take a large number of trajectories before the state-action space is explored adequately.

- **ϵ-greedy** Pick a value ϵ between 0 and 1. With probability $1 - \epsilon$, use the best known action. With probability ϵ, choose an action randomly. The idea is that areas thought to yield high reward receive the most attention, but occasionally choosing random actions ensures adequate exploration over time. ϵ may be treated as a constant or as decreasing function of the number of transitions observed. This latter variant will be seen in Chapter 2.

- **Optimism in the face of uncertainty** Always choose the best known action, but initialize $\widehat{Q}_0(s, a)$ to a value higher than the maximum possible reward. This encourages the agent to "optimistically" try unknown actions until estimated state-action values begin to converge.

- **Quasi-random exploration** Most commonly $s_{n+1} = u_n$. That is, the beginning state for the next iteration is the state at which the system ends in the current iteration. This creates data consisting of unbroken chains of states or trajectories. Even though a researcher may have a model that can simulate transitions from arbitrary states, it is customary to always use the previous ending state unless some condition is met that forces a return to an initial

state. This is so exploration mimics the collection of data from a physical, online system which cannot arbitrarily change states. If one is willing to depart from custom, uniform exploration can be achieved by choosing s_{n+1} according to a low-discrepancy or quasi-random sequence. See Appendix A for details on quasi-random sequences.

1.4 What Constitutes a Solution

In practice, a researcher will use Reinforcement Learning to obtain a policy that can successfully carry out a task, where "successfully" is measured in an intuitive or qualitative sense. For example, a vacuum cleaning robot measures battery level and distance from its recharge station as part of its state, and is successful if it can consistently keep itself powered while cleaning.

In order to learn a good policy, we cast the problem quantitatively as an optimization problem with Equation 1.1 (or equivalently, Equation 1.2) as the objective. Solution methods such as Q-learning or policy iteration are designed to try and solve this optimization problem and have proofs that their policies converge to optimal policies. However, when evaluating whether a policy is good enough, researchers are typically less rigorous. Policies are not evaluated based on how well the optimization is solved, for two reasons. For one, actually verifying that the maximum value is attained is infeasibly difficult for many non-trivial MDPs. Second, the researcher is likely to view and treat the solution method heuristically, and consider the problem solved when a policy is observed to qualitatively complete the task satisfactorily. Another consideration is that non-trivial problems will take significant computation to obtain policies, and a near-optimal policy will often be preferred over a marginally better one that takes much longer to obtain.

It is in this light that the solutions obtained in Chapters 2 and 3 should be viewed. It is desired and worthwhile to prove that the algorithms yield asymptotically optimal policies, but determing when an application has been solved adequately will likely dispense with some of the rigor.

Chapter 2

Convergence of a Continuous Algorithm

The previous chapter introduced Q-learning, a novel and popular algorithm for learning the value of a state-action pair under an optimal policy without explicitly estimating transition probabilities and mean rewards. The values can then be used to construct an optimal policy. If there are a finite number of states and actions, then the learned values are proven to converge to their true values. However, problems with a very large number of states or actions may cause the values to converge unreasonably slowly. Furthermore, if the number of state-action pairs is infinite, then the convergence guarantee no longer holds. A natural question is whether the knowledge gained from an observation can be generalized using function approximators to similar states and actions, and whether the state-action values still converge, either theoretically or empirically. The immediacy of this question is apparent even in Watkins' thesis [36], where a form of function approximation, the CMAC [1], is used in the solution to two example problems with a large number of state-action pairs.

Function approximators in reinforcement learning may cause value estimates to converge to suboptimal values [3], oscillate [11], or even diverge [9] [35]. However, when used carefully, they have also been impressively effective. A famous example of success with a large yet finite state-action space is the TD-Gammon program [33], which used temporal difference learning [27] and a neural network to generalize knowledge. It eventually learned to be competitive with tournament-level

human backgammon players. Scenarios with continuous states and finite actions have been well studied. Empirical convergence has been observed with CMACs [28], neural networks [23] [32], and regression trees [8]. Theoretically, there have been positive results as well. A parametric function approximator can yield convergent results if it satisfies certain interpolation properties [30]. A non-parametric technique, kernel regression, has also been proven to converge to optimal values [21].

Problems where both the states and the actions are allowed to be continuous have been less well-studied. One major reason is that constructing a policy requires taking a supremum over all possible actions, which is straightforward when actions are finite but generally difficult when continuous. This difficulty, though present, has not prevented researchers from developing techniques to make the process manageable. One method is to calculate values for a discrete set of actions, then use a weighted average to produce a continuous-valued action [18]. Baird & Klopf [2] specifically designed a function approximator, termed "wire-fitting," such that the supremum over the action space can be immediately obtained. Wire-fitting has been combined with neural networks to learn thruster-control in order to drive a submersible to a target with success [10]. Algorithms allowing for stochastic policies can rapidly choose actions from a probability distribution, as in Sequential Monte Carlo Learning [17].

Despite the practical success of these algorithms, there has been very little theoretical investigation into the convergence properties when actions are continuous. This chapter presents what is, to my knowledge, the first off-policy Q-learning variant allowing for continuous state and actions which is proven to converge to optimal values with probability one. Specifically, if the state and action spaces are both allowed to be infinite but compact subsets of Euclidean space, then with a sufficiently small kernel regression bandwidth and suitable continuity conditions on the random rewards and transition probabilities, one may obtain a Q-value function estimate that is uniformly within an arbitrary tolerance of the true Q-value function. Though the intention of the algorithm is to fill a theoretical gap, the results of a proof-of-concept example implementation will be provided.

The particular method of knowledge generalization is Nadaraya-Watson kernel regression, which is a memory-based non-parametric smoothing technique with well-studied properties. It is similar to the function approximators used by Ormoneit & Sen [21] and Santamaría et al. [24]. To avoid confusion, a distinction should be made between Nadaraya-Watson kernel regression and another technique based on kernel functions that has been applied to reinforcement learning, commonly known as kernelized regression or simply "the kernel trick." This method uses a kernel function to

obtain the result of an inner product calculation in a higher-dimensional, feature-rich space while avoiding some of the computational cost of mapping the vectors into the higher-dimensional space. The inner product calculation is used to perform a linear regression in the higher-dimensional space, resulting in a non-linear regression in the original space. Nadaraya-Watson kernel regression and kernelized regression are similar in that they use a kernel function and deliver non-linear regression results, but the intermediate methodologies are quite different. See Taylor & Parr [31] for a discussion of kernelized value function methods, and Xu et. al. [39] for an application of kernelized regression to policy iteration.

In Section 2.1, Markov Decision Processes will be defined. Section 2.2 introduces Nadaraya-Watson kernel regression and describes an algorithm that uses kernel smoothing to obtain a function that estimates the Q-value of each state-action pair. The main result of this paper, theoretical convergence of a continuous state and action algorithm, is stated in Section 2.3 along with sufficient conditions for convergence. Section 2.4 describes the strategy of the proof, proves a set of lemmas, and finally proves the main result. An example implementation along with a discussion of practical difficulties is in Section 2.5. Finally, Section 2.6 discusses a few ideas for extensions.

2.1 Description of a Continuous Markov Decision Process

Subsection 1.1 defined discrete Markov Decision Processes. Mirroring that, this section will consider Markov Decision Processes for which the state and action spaces are not discrete, but are compact subsets of Euclidean space. Let S, the state space, be a compact subset of Euclidean space of dimension d_S and $\mathcal{B}(S)$ be the Borel σ-algebra on S. Let A, the action space, be a compact subset of Euclidean space of dimension d_A. Note that $S \times A$ is a subset of $\mathbb{R}^d$ where $d = d_S + d_A$. Elements of this space will be written (s, a) to make the state and action clear, but for all computational purposes they are best regarded as numerical vectors. Transitions between states are governed by a function $P : S \times \mathcal{B}(S) \times A \to \mathbb{R}_+$ satisfying:

- For each $a \in A$ and $B \in \mathcal{B}(S)$, $P(\cdot, B, a) : S \to \mathbb{R}_+$ is a measurable function.
- For each $s \in S$ and $a \in A$, $P(s, \cdot, a) : \mathcal{B}(S) \to \mathbb{R}_+$ is a probability measure on S.
- For each $s \in S$ and $B \in \mathcal{B}(S)$, $P(s, B, \cdot) : A \to \mathbb{R}_+$ is a measurable function.

For each state and action (s, a) there is a random reward $R(s, a)$ bounded by a constant C_0.

The Markov Decision Process proceeds as follows: the process begins in some state $s \in S$ and the controller chooses an action $a \in A$. A random reward $R(s, a)$ is received and the system transitions to a new state in accordance with the probability measure $P(s, \cdot, a)$. The system progresses to the next time step, and this scenario repeats from the new state. The reward distributions and transition probabilities are assumed to have the Markov property: they only depend on the current state and action; they do not depend on the history of previous states and actions.

A policy π is a measurable function $\pi : S \to A$. In some settings, policies are allowed to be stochastic or non-stationary, but we will restrict our attention to deterministic policies. Define $P_\pi(s, B) := P(s, B, \pi(s))$. Then P_π is a transition kernel for a Markov chain on the space S.

Let $\gamma, 0 < \gamma < 1$, be a discount factor. Let (s_t, a_t) be the (random) state and action at time t. Under a policy π, the value of a state s is defined under the expected total discounted reward criterion:

$$V^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t)) | s_0 = s\right].$$

A policy π^* is optimal if it satisfies $V^{\pi^*}(s) = \sup_\pi V^\pi(s), \forall s \in S$.

Our goal is to find an algorithm which will, under suitable conditions, converge to an optimal policy. To that end, we will define *Q-values* as

$$Q^\pi(s, a) := E[R(s, a)] + \gamma \int_S V^\pi(t) P(s, dt, a).$$

This is a continuous version of the discrete definition made by Watkins. Intuitively $Q^\pi(s, a)$ is the value obtained if we start in state s, utilize action a, and then follow policy π thereafter. Consider the Q-values associated with an optimal policy π^*: $Q^*(s, a) := Q^{\pi^*}(s, a)$. If we can learn the values $Q^*(s, a)$ for all (s, a), then an optimal policy can easily be recovered by setting

$$\pi^*(s) = \operatorname*{argsup}_{a \in A} Q^*(s, a).$$

2.2 Description of Algorithm

This section will introduce the form of the function approximator to be used, and then describe an algorithm which will generate a sequence of estimates of the optimal Q-values.

2.2.1 Nadaraya-Watson Kernel Regression

Nadaraya-Watson kernel regression [38] [20] is a smoothing technique which estimates the value at a point as a weighted average of nearby observations, using a non-negative kernel function to assign weights to observations based on their distance. Kernels typically are symmetric, peak at zero, and decrease away from zero so that the weight of an observation is inversely related to the distance. Formally, we will require $K : \mathbb{R}^d \to \mathbb{R}_+$ to be a multivariate function satisfying:

KI. K is integrable: $\int_{\mathbb{R}^d} K(u)du < \infty$.

KII. K is bounded: $||K||_\infty < C_K < \infty$.

KIII. K has compact support: There exists $L < \infty$ such that

$$K(u) = 0 \text{ when } ||u||_\infty > L.$$

KIV. K is Lipschitz: There exists $M < \infty$ such that for all $u, u' \in \mathbb{R}^d$,

$$|K(u) - K(u')| \leq M||u - u'||_\infty.$$

The compact support condition is not required in all applications (indeed, the Gaussian function is a common kernel choice) but will be used here to reduce computation load by assigning zero weight to sufficiently distant observations.

A bandwidth $h > 0$ is a parameter used to fine-tune the level of smoothing. Define

$$K_h(u) := \frac{1}{h}K(u/h).$$

If u is non-scalar, the division may be understood to be component-wise. Values of h which are too large or small relative to the sample size tend to oversmooth or overfit the data. Typically when using kernel regression the bandwidth decreases toward zero as the number of observations increases. A discussion of the particulars of bandwidth selection is beyond the scope of this chapter, but a rule of thumb obtained from assuming Gaussian density and unit variance is $h = n^{-\frac{1}{4+d}}$, where n is the number of observations and d is the dimension [13]. This rule of thumb should be regarded as a starting value, not a definitive answer. For example, cross-validation is a common method

for selecting bandwidths. For more information, the interested reader is referred to the texts by Silverman [25] or Simonoff [26]. For the purpose of the algorithm to follow, we need only a constant bandwidth that is sufficiently small.

Given a set of n ordered pairs (x_i, y_i) and a bandwidth h, the Nadaraya-Watson estimator is

$$\widehat{m_h}(x) = \frac{\sum_{i=1}^{n} K_h(x - x_i) y_i}{\sum_{i=1}^{n} K_h(x - x_i)}.$$

2.2.2 The Algorithm

We will use the following notation. For $n > 0$,

- h is a fixed bandwidth.
- γ is a fixed discount factor.
- (s_n, a_n) is the state-action pair at the beginning of iteration n.
- r_n is the sample reward observed at iteration n.
- u_n is the state that is transitioned to during iteration n.
- $y_{h,n} := r_n + \gamma \sup_{a \in A} \widehat{Q}_{h,n-1}(u_n, a)$, where $\widehat{Q}_{h,n-1}(u_n, a)$ is defined below in (2.2).

Here is a description of the algorithm.

1. Set the initial Q-value estimate function to be zero everywhere. $\widehat{Q}_{h,0}(s, a) = 0$ for all $(s, a) \in S \times A$. Choose some initial state s_1.

2. For $n > 0$, pick an action a_n ϵ-greedily (or according to some other exploration method). Define a function $\alpha_{h,n} : S \times A \to [0, 1]$ by

$$\alpha_{h,n}(s, a) = \begin{cases} \dfrac{K_h((s, a) - (s_n, a_n))}{\sum_{j=1}^{n} K_h((s, a) - (s_j, a_j))} & \text{if } \sum_{j=1}^{n} K_h((s, a) - (s_j, a_j)) \neq 0 \\ 0 & \text{if } \sum_{j=1}^{n} K_h((s, a) - (s_j, a_j)) = 0 \end{cases} \tag{2.1}$$

   and use this to update the Q-value estimate function by setting

$$\widehat{Q}_{h,n}(s, a) := (1 - \alpha_{h,n}(s, a))\widehat{Q}_{h,n-1}(s, a) + \alpha_{h,n}(s, a) y_{h,n}. \tag{2.2}$$

Set $s_{n+1} = u_n$.

Note that as a result of Equation (2.2), one may show by induction that

$$\widehat{Q}_{h,n}(s,a) = \begin{cases} \dfrac{\sum_{j=1}^{n} K_h((s,a)-(s_j,a_j))y_{h,j}}{\sum_{j=1}^{n} K_h((s,a)-(s_j,a_j))} & \text{if } \sum_{j=1}^{n} K_h((s,a)-(s_j,a_j)) \neq 0 \\ \widehat{Q}_{h,0}(s,a) & \text{if } \sum_{j=1}^{n} K_h((s,a)-(s_j,a_j)) = 0. \end{cases} \tag{2.3}$$

Equation (2.2) is how the updates are conceptually performed and has a form familiar to classic Q-learning, but Equation (2.3) is how all calculations are made. The estimates produced by this algorithm are essentially kernel-smoothed averages of the terms $y_{h,n} := r_n + \gamma \sup_{a \in A} \widehat{Q}_{h,n-1}(u_n, a)$.

Algorithm 1 Pseudocode for theoretical algorithm

```
Initialize h = bandwidth value, m = maximum iterations,
γ = discount factor, ε = exploration parameter
Initialize Q̂_{h,0}(s,a) = 0  ∀(s,a)
Set initial state s_1
for i=1:m do
    r = Uniform(0,1) random value
    if r < ε then a_i = random action
    else
        a_i = sup_{a∈A} Q̂_{h,n-1}(s_i, a)
    end if
    u_i = next state, r_i = reward
    y_{h,i} := r_i + γ sup_{a∈A} Q̂_{h,i-1}(u_i, a)
    Q̂_{h,i}(s,a) = (Σ_{j=1}^{i} K_h((s,a)-(s_j,a_j)) y_{h,j}) / (Σ_{j=1}^{i} K_h((s,a)-(s_j,a_j)))
    s_{i+1} = u_i
end for
```

2.3 Statement of Theorem

Assume the following conditions hold:

AI. The state-action pair to be utilized at the beginning of an iteration (regarded as a random variable due to exploration and random transitions) has a density $f : S \times A \to R$ which is positive everywhere and has uniformly continuous and bounded second derivatives.

AII. Rewards are bounded. There exists a $C_0 < \infty$ such that for any $(s,a) \in S \times A$, $R(s,a) < C_0$.

AIII. The expected values of the rewards are Lipschitz continuous across the state-action space. There exists a C_r such that for all $(s_1, a_1), (s_2, a_2) \in S \times A$,

$$|E[R(s_1, a_1)] - E[R(s_2, a_2)]| \leq C_r ||(s_1, a_1) - (s_2, a_2)||.$$

Also, $E[R(s, a)]f(s, a)$ has uniformly continuous and bounded second derivatives.

AIV. Transition probabilities converge weakly and Lipschitz continuously. Let $g : S \to \mathbb{R}$ be continuous and bounded. There exists a C_t such that for any $(s_1, a_1), (s_2, a_2) \in S \times A$,

$$\left| \int g(u)P(s_1, du, a_1) - \int g(u)P(s_2, du, a_2) \right| \leq C_t ||g(u)||_\infty ||(s_1, a_1) - (s_2, a_2)||$$

Also, $\int g(u)P(s, du, a)f(s, a)$ has uniformly continuous and bounded second derivatives.

Theorem 2. *Let assumptions AI. - AIV. and kernel conditions KI. - KIV. hold and $\widehat{Q}_{h,n}(s, a)$ be defined by (2.2). With probability one, for any $\epsilon > 0$, there exists $h = h(\epsilon)$ and $N = N(h, \epsilon)$ such that for $n > N$*

$$\sup_{(s,a) \in S \times A} |\widehat{Q}_{h,n}(s, a) - Q^*(s, a)| < \epsilon. \tag{2.4}$$

2.4 Lemmas and Proof of Theorem

The proof strategy is inspired by Watkins' original proof over finite spaces. For the original proof, we suggest the technical report [37] that followed Watkins' thesis. This method is different from the later approaches to proving convergence of Q-learning, such as those of Tsitsiklis [34] and Jaakkola et al. [14], which are variations and extensions of stochastic approximation theory. Although Watkins' method does rely on a stochastic approximation result in a minor way, the key intuition is how Q-learning imitates model estimation.

One of the strengths of Q-learning is that it does not require a model of the system to be estimated or maintained. However, the learned Q-values are the optimal values for a model that can be constructed by appropriately assigning weights to the observed rewards and transitions. Specifically, if the learning parameters used to update value estimates are used to weight reward and transition observations to carefully define an artificial process, then the learned Q-value estimates will be optimal for this artificial process. As more rewards and transitions are observed, the artificial

process becomes more similar to the real process, so the optimal Q-values for the two processes become more similar, thus the learned values converge to the optimal values for the real process. Watkins called such an artificial process the Action Replay Process.

The proof strategy can be summarized in the following five steps:

1. For a fixed bandwidth value, define an auxiliary Action Replay Process (ARP). This artificial process is purely a proof device.

2. Show that the function $\widehat{Q}_{h,n}(s,a)$ is the optimal Q-value function for the ARP with corresponding bandwidth.

3. Show that for sufficiently small bandwidths, the rewards and transition probabilities from the ARP become arbitrarily close to those of the original process.

4. Show that two MDPs with similar rewards and transition probabilities have similar optimal Q-values.

5. The optimal values of the ARP converge to the optimal values of the original process. Hence by step 2, the Q-values learned by the algorithm converge to the optimal Q-values.

2.4.1 Construction of the ARP (Action Replay Process)

Now the ARP will be defined. It is a finite length, terminating MDP. Pick a bandwidth value h, and keep it fixed throughout the discussion that follows. We suggest thinking of the ARP as a card game. Suppose that one is preparing to run the algorithm from Section 2.2. Also suppose that we have a deck of index cards. For each iteration of the algorithm, we will write the following values on a card: $< s_k, a_k, u_k, r_k, \widehat{Q}_{h,k}(\cdot), \alpha_{h,k}(\cdot) >$. These values are regarded as random variables depending on the random states, actions, and rewards from the original process rather than fixed values from a particular sample trajectory. We also have a card on which the initial $\widehat{Q}_{h,0}(\cdot)$ is written. With this card on bottom, stack all cards in ascending order by iteration as shown in Figure 2.1.

States A state of the ARP is a 2-tuple $< s, n >$ consisting of a state from the original state space $s \in S$ and a non-negative level n that tells us which card is to be inspected. All cards above level n are ignored, leaving us with a finite stack of cards. Any state with level 0 is a terminal, absorbing state.

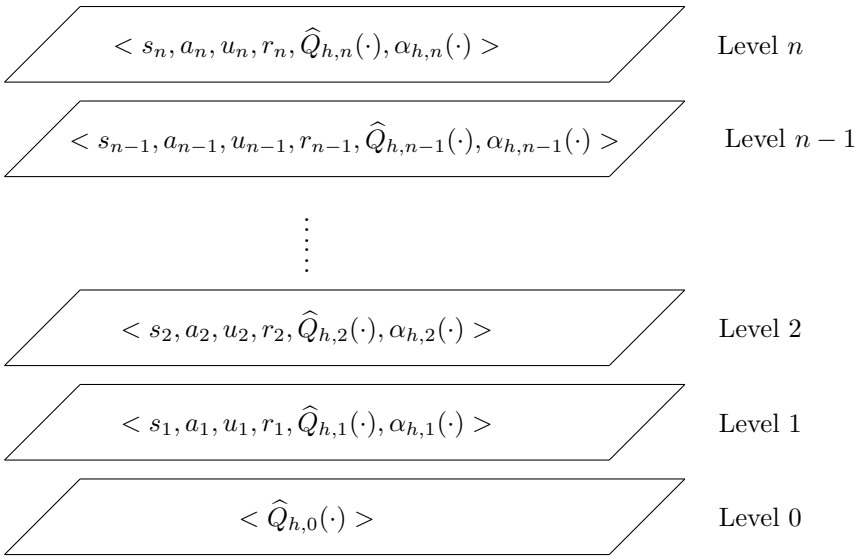

Figure 2.1: The ARP envisioned as a stack of cards.

Actions The actions for the ARP are the same as the actions from the real process, $a \in A$.

Transitions and Rewards Suppose the ARP is in state $< s, n >$ and action a is chosen. Inspect the card corresponding to level n. With probability $\alpha_{h,n}(s, a)$, we accept this card and receive reward r_n and transition to state $< u_n, n-1 >$. Otherwise, ignore that card and inspect the card at level $n-1$. With probability $\alpha_{h,n-1}(s, a)$, accept this card, receive reward r_{n-1} and transition to state $< u_{n-1}, n-2 >$. Otherwise, continue inspecting cards until one is accepted, where the probability of accepting the card at level k is $\alpha_{h,k}(s, a)$. If the bottom card corresponding to level 0 is reached (that is, the state is $< s, 0 >$ and action a is being utilized) then a reward $Q_{h,0}(s, a)$ is received and the ARP terminates.

As the ARP is an MDP, it has optimal Q-values. Let $Q_h^*(< s, n >, a)$ be the function giving the optimal Q-values for state $< s, n >$ and action a for the ARP with bandwidth h.

2.4.2 Lemmas

The optimal Q-values at level n for the ARP corresponding to bandwidth h are the learned Q-value estimates for the original process at iteration n as defined by (2.2).

Lemma 1.

$$Q_h^*(< s, n >, a) = \widehat{Q}_{h,n}(s, a), \forall (s, a) \in S \times A, \forall n \geq 0.$$

Proof. Proceed by induction on the level n of the ARP. Recall that if the ARP is in state $< s, 0 >$ and action a is used, then the reward is $\widehat{Q}_{h,0}(s,a)$ and the process terminates. Thus

$$Q_h^*(< s, 0 >, a) = \widehat{Q}_{h,0}(s,a)$$

which proves the $n = 0$ case.

Now suppose by way of induction that $Q_h^*(< s, n-1 >, a) = \widehat{Q}_{h,n-1}(s,a)$ for all states and actions. Let the ARP be in state $< s, n >$ with action a utilized. Recall that by construction of the ARP,

- with probability $\alpha_{h,n}(s,a)$ we receive reward r_n and transition to $< u_n, n-1 >$. Otherwise,
- with probability $1-\alpha_{h,n}(s,a)$ that card is thrown away and the situation is identical to utilizing a in state $< s, n-1 >$.

Then by conditioning on whether the level n card is accepted or not, we have by the induction hypothesis and (2.2):

$$\begin{aligned}
Q_h^*(< s, n >, a) &= (1 - \alpha_{h,n}(s,a)) Q_h^*(< s, n-1 >, a) \\
&\quad + \alpha_{h,n}(s,a)(r_n + \gamma \sup_{b \in A} Q_h^*(< u_n, n-1 >, b)) \\
&= (1 - \alpha_{h,n}(s,a)) \widehat{Q}_{h,n-1}(s,a) \\
&\quad + \alpha_{h,n}(s,a)(r_n + \gamma \sup_{b \in A} \widehat{Q}_{h,n-1}(u_n, b)) \\
&= (1 - \alpha_{h,n}(s,a)) \widehat{Q}_{h,n-1}(s,a) + \alpha_{h,n}(s,a) y_{h,n} \\
&= \widehat{Q}_{h,n}(s,a).
\end{aligned}$$

□

The following lemma, which is needed in the proof of Lemma 3, shows that $\sum_{k=1}^{n} \alpha_{h,k}(s,a)$ grows uniformly across the state-action space.

Lemma 2. *If assumption AI. holds, then with probability one,*

$$\lim_{n \to \infty} \inf_{(s,a) \in S \times A} \sum_{k=1}^{n} \alpha_{h,k}(s,a) \to \infty. \tag{2.5}$$

Proof. $S \times A$ is compact, so it can be covered by a finite number of disjoint sets of fixed positive diameter with each set having positive Lebesgue measure. Choose the diameter such that u, u' in the same set implies $K_h(u - u') > \delta > 0$. Suppose that J sets are needed, and denote them by $B_1, B_2, ..., B_J$. Set $T_0 = 0$ and for $k > 0$, let T_k denote the first time at which each set has been visited at least k times. Note that by AI. and the Borel-Cantelli Lemma, $P(T_k < \infty) = 1$. Fix $(s, a) \in S \times A$, and letting C_K denote the bound on the kernel values,

$$\sum_{k=1}^{T_n} \alpha_{h,k}(s,a) = \sum_{k=1}^{n} \sum_{j=T_{k-1}+1}^{T_k} \alpha_{h,j}(s,a) \geq \sum_{k=1}^{n} \frac{\delta}{C_K T_k} = \frac{\delta}{C_K} \sum_{k=1}^{n} T_k^{-1}.$$

The terms $T_k, k > 0$, are not independent but can be bounded by random variables that are. Set $W_1 = T_1$. Ignoring all previous visits to sets prior to and at time T_1, set W_2 to be the number of steps from T_1 until each B_j is visited again. It is obvious that $W_2 \geq T_2 - T_1$. Likewise, for $k > 1$, set W_k to be the number of steps from T_{k-1} until each set is visited again, so that $W_k \geq T_k - T_{k-1}$. Notice that $W_1, W_2, ...$ are i.i.d. random variables. Applying these inequalities, one may write

$$\begin{aligned} T_1^{-1} &= W_1^{-1} \\ T_2^{-1} &\geq (W_1 + W_2)^{-1} \\ &\vdots \\ T_k^{-1} &\geq \left(\sum_{j=1}^{k} W_j \right)^{-1}. \end{aligned}$$

We now have

$$\sum_{k=1}^{T_n} \alpha_{h,k}(s,a) \geq \frac{\delta}{C_K} \sum_{k=1}^{n} \left(\sum_{j=1}^{k} W_j \right)^{-1}.$$

It suffices to show that $\sum_{k=1}^{n} \left(\sum_{j=1}^{k} W_j \right)^{-1}$ goes to infinity with probability one. Set $S_n = \sum_{k=1}^{n} W_k$. Then $\sum_{k=1}^{n} \frac{1}{S_k} = \sum_{k=1}^{n} \frac{1}{k} \frac{k}{S_k}$. Since $\frac{k}{S_k} \to \frac{1}{E[W_1]} > 0$, $\sum_{k=1}^{n} \frac{1}{k} \frac{k}{S_k} \to \infty$ with probability one. Since (s, a) was arbitrary, the result is proved. □

The next lemma shows that if one starts the ARP at a sufficiently high level, the probability of ending below a certain level after a fixed number of actions can be made arbitrarily small.

Lemma 3. *Let $\epsilon > 0$, l_0 be any level of the ARP, and positive integer n represent the length of a sequence of actions to be followed. Then there exists a level $l > l_0$ such that if the ARP starts at a level above l and follows the sequence of n actions, the probability of ending below level l_0 is less than ϵ.*

Proof. Proceed by induction. Consider first the case where $n = 1$. For $m > l_0$, suppose the ARP is in state $< s, m >$ and action a is utilized. Transitioning to a state with level below l_0 means no card at or above level l_0 is accepted. The probability of not accepting the card at level k is $1 - \alpha_{h,k}(s, a)$, so the probability of accepting none is $\Pi_{k=l_0}^{m}(1 - \alpha_{h,k}(s, a))$. Using $1 - x \leq e - x$ yields

$$\Pi_{k=l_0}^{m}(1 - \alpha_{h,k}(s, a)) < e^{-\sum_{k=l_0}^{m} \alpha_{h,k}(s,a)}.$$

It follows from Lemma 2 that it is possible to choose l_1 such that $m > l_1$ implies

$$\inf_{(s,a)\in S\times A} \sum_{k=l_0}^{m} \alpha_{h,k}(s, a) > -\log \epsilon.$$

Then

$$P(\text{ending below } l_0) = \Pi_{k=l_0}^{m}(1 - \alpha_{h,k}(s, a)) < e^{-\sum_{k=l_0}^{m} \alpha_{h,k}(s,a)} < \epsilon.$$

For a sequence of n actions, there exists l_1 such that the probability of ending below l_0 from l_1 in one transition is less than $\frac{\epsilon}{2}$ and by induction there is an l_2 such that the probability of ending below l_1 in $n - 1$ transitions is less than $\frac{\epsilon}{2}$. Then if the ARP starts above level l_2,

$$\begin{aligned}
&P(\text{below } l_0 \text{ after } n \text{ transitions}) \\
=&P(\text{below } l_0 \text{ after } n \text{ transitions}|\text{above } l_1 \text{ after } n-1 \text{ transitions}) \\
&* P(\text{above } l_1 \text{ after } n-1 \text{ transitions}) \\
+&P(\text{below } l_0 \text{ after } n \text{ transitions}|\text{below } l_1 \text{ after } n-1 \text{ transitions}) \\
&* P(\text{below } l_1 \text{ after } n-1 \text{ transitions}) \\
\leq&\frac{\epsilon}{2} \cdot 1 + 1 \cdot \frac{\epsilon}{2} = \epsilon.
\end{aligned}$$

□

The next lemma allows one to work with finite sequences of actions rather than infinite ones

with an arbitrarily small error.

Lemma 4. *Consider the value of a state s when a specific sequence of n actions $(a_0, a_1, ..., a_{n-1})$ is to be followed and the process is then terminated:*

$$E\left[\sum_{t=0}^{n-1}\gamma^t R(s_t, a_t)|s_0 = s\right].$$

Also consider the value of the same state under the same n actions but then followed by an arbitrary policy π:

$$E\left[\sum_{t=0}^{n-1}\gamma^t R(s_t, a_t) + \sum_{t=n}^{\infty}\gamma^t R(s_t, \pi(s_t))|s_0 = s\right].$$

The difference of these two values goes to zero as n increases towards infinity.

Proof. We are interested in the difference

$$E\left[\sum_{t=0}^{n-1}\gamma^t R(s_t, a_t) + \sum_{t=n}^{\infty}\gamma^t R(s_t, \pi(s_t))|s_0 = s\right] - E\left[\sum_{t=0}^{n-1}\gamma^t R(s_t, a_t)|s_0 = s\right].$$

The difference is clearly the expectation of the second term in the first expectation. Because the rewards are bounded by C_0, we can use a change of variables $v = t - n$ and write

$$\begin{aligned} E\left[\sum_{t=n}^{\infty}\gamma^t R(s_t, \pi(s_t))|s_0 = s\right] &= E\left[\sum_{v=0}^{\infty}\gamma^{v+n} R(s_v, \pi(s_v))|s_0 = s\right] \\ &\leq \gamma^n E\left[\sum_{v=0}^{\infty}\gamma^v |R(s_v, \pi(s_v))||s_0 = s\right] \\ &\leq \gamma^n \sum_{v=0}^{\infty}\gamma^v C_0 = \gamma^n \frac{C_0}{1-\gamma} \end{aligned}$$

which goes to zero as n goes to infinity. □

The next lemma will consider the reward received from the ARP when in state $< s, n >$ and action a is utilized. This random quantity depends on the state-actions chosen and rewards received in the original process, but it also depends on which levels of the ARP are accepted during state transitions. For the ARP constructed with bandwidth h and rewards received when in state $< s, n >$ and utilizing action a, let $\widehat{E}[R_{h,n}(s, a)]$ denote an expectation taken over ARP state transitions, but not over the sequence of state-actions and rewards from the original process. Note that $\widehat{E}[R_{h,n}(s, a)]$ is not a constant, but a random quantity defined on the same sample space as the original process.

Similarly, the probability of the ARP transitioning into a state in set $T \in \mathcal{B}(S)$ is also a random variable depending on which state-actions are chosen in the original process. For the ARP constructed with bandwidth h and starting in state $< s, n >$ and utilizing action a, let $\widehat{P}_{h,n}(s, T, a)$ denote the random probability of the ARP transitioning into a state in set T.

The next lemma shows that for high enough levels, the rewards and transition probabilities of the ARP are close to the rewards and transition probabilities of the original process.

Lemma 5. *If assumptions AI.-AIV. and kernel conditions KI.-KIV. are met, then*

a) With probability one, for any $\epsilon > 0$ there exists an $h = h(\epsilon, T)$ and $N = N(\epsilon, h)$ such that if $n > N$,

$$\sup_{(s,a)\in S\times A} \left| \widehat{E}[R_{h,n}(s,a)] - E[R(s,a)] \right| < \epsilon.$$

b) With probability one, for any $\epsilon > 0$ and $g : S \to \mathbb{R}$ that is integrable, continuous, and bounded, there exists a $h(\epsilon)$ and $N(\epsilon, h)$ such that if $n > N$,

$$\sup_{(s,a)\in S\times A} \left| \int_S g(u)\widehat{P}_{h,n}(s, du, a) - \int_S g(u)P(s, du, a) \right| < \epsilon.$$

Proof. Consider the expected reward associated with a state-action (s, a) at level n of the ARP. Condition on whether the card at level n is accepted. With probability $1 - \alpha_{h,n}(s, a)$ we do not accept the card, and the expected reward is that of level $n-1$. With probability $\alpha_{h,n}(s, a)$ the level n card is accepted and we receive reward r_n. For the special case $n = 0$, we are at the lowest level of the ARP and receive reward $\widehat{Q}_{h,0}(s, a)$. So the expected rewards satisfy the recursive relationship

$$\widehat{E}[R_{h,0}(s,a)] = \widehat{Q}_{h,0}(s,a),$$
$$\widehat{E}[R_{h,n}(s,a)] = (1 - \alpha_{h,n}(s,a))E[R_{h,n-1}(s,a)] + \alpha_{h,n}(s,a)r_n, \text{ for } n > 0.$$

Now we show by induction that

$$\widehat{E}[R_{h,n}(s,a)] = \frac{\sum_{j=1}^{n} K_h((s,a) - (s_j, a_j))r_j}{\sum_{j=1}^{n} K_h((s,a) - (s_j, a_j))}$$

when $\sum_{j=1}^{n} K_h((s,a)-(s_j,a_j)) \neq 0$. For the $n=1$ case,

$$\widehat{E}[R_{h,1}(s,a)] = (1-\alpha_{h,1}(s,a))\widehat{Q}_{h,0}(s,a) + \alpha_{h,1} r_1.$$

Replace the $\alpha_{h,1}(s,a)$ terms as per the definition in (2.1).

$$\begin{aligned}\widehat{E}[R_{h,1}(s,a)] &= \left(1 - \frac{\sum_{j=1}^{1} K_h((s,a)-(s_j,a_j))}{\sum_{j=1}^{1} K_h((s,a)-(s_j,a_j))}\right)\widehat{Q}_{h,0}(s,a) + \frac{\sum_{j=1}^{1} K_h((s,a)-(s_j,a_j))}{\sum_{j=1}^{1} K_h((s,a)-(s_j,a_j))} r_1 \\ &= r_1.\end{aligned}$$

Now assume the induction hypothesis holds for $n-1$.

$$\begin{aligned}\widehat{E}[R_{h,n}(s,a)] &= (1-\alpha_{h,n}(s,a))\widehat{E}[R_{h,n-1}(s,a)] + \alpha_{h,n}(s,a) r_n \\ &= \left(\frac{\sum_{j=1}^{n-1} K_h((s,a)-(s_j,a_j))}{\sum_{j=1}^{n} K_h((s,a)-(s_j,a_j))}\right)\left(\frac{\sum_{j=1}^{n-1} K_h((s,a)-(s_j,a_j)) r_j}{\sum_{j=1}^{n-1} K_h((s,a)-(s_j,a_j))}\right) \\ &+ \frac{K_h((s,a)-(s_j,a_j))}{\sum_{j=1}^{n} K_h((s,a)-(s_j,a_j))} r_n \\ &= \frac{\sum_{j=1}^{n} K_h((s,a)-(s_j,a_j)) r_j}{\sum_{j=1}^{n} K_h((s,a)-(s_j,a_j))}.\end{aligned}$$

The expected rewards for the ARP are equivalent to kernel regression estimates. Assumptions AI., AII., AIII., and kernel conditions KI.-KIV. meet the technical conditions of Hansen [12]: see Theorem 9 on page 735. This yields

$$\sup_{(s,a)\in S\times A} |\widehat{E}[R_{h,n}(s,a)] - E[R(s,a)]| < \epsilon$$

for sufficiently large n, which proves part a).

For part b), recall that u_n denotes the state that is transitioned to at iteration n. Consider the random variable indicator function for the event that the nth transition from the original process is into $T \in \mathcal{B}(S)$.

$$1_T(u_n) = \begin{cases} 1 & \text{if } u_n \in T \\ 0 & \text{if } u_n \notin T \end{cases}.$$

Similar to rewards, the transition probabilities for the ARP at level n can be conditioned on whether

or not the level n card is accepted. Assume we are in state-action (s,a). With probability $1-\alpha_{h,n}(s,a)$, we do not accept the card, and the probability of transitioning into T is that of level $n-1$. With probability $\alpha_{h,n}(s,a)$, the level n card is accepted and the value of $1_T(u_n)$ tells us whether we transition into T or not. Thus we have the relation

$$\begin{aligned}\widehat{P}_{h,1}(s,T,a) =&\alpha_{h,1}(s,a)1_T(u_1),\\ \widehat{P}_{h,n}(s,T,a) =&(1-\alpha_{h,n}(s,a))\widehat{P}_{h,n-1}(s,T,a)+\alpha_{h,n}(s,a)1_T(u_n).\end{aligned}$$

Let $g: S\to\mathbb{R}$ be a simple function, $g(u)=\sum_{i=1}^m a_i 1_{T_i}(u)$. We can apply this relation to the integral of g to find

$$\begin{aligned}\int g(u)\widehat{P}_{h,1}(s,du,a) &= \sum_{i=1}^m a_i\widehat{P}_{h,1}(s,T_i,a)=\sum_{i=1}^m a_i 1_{T_i}(u_1)\\ &= g(u_1)\end{aligned}$$

and

$$\begin{aligned}\int g(u)\widehat{P}_{h,n}(s,du,a) &= \sum_{i=1}^m a_i\widehat{P}_{h,1}(s,T_i,a)\\ &= \sum_{j=1}^m a_i\left[(1-\alpha_{h,n}(s,a))\widehat{P}_{h,n-1}(s,T_i,a)+\alpha_{h,n-1}(s,a)1_{T_i}(u_n)\right]\\ &= (1-\alpha_{h,n-1}(s,a))\int g(u)\widehat{P}_{h,n-1}(s,du,a)+\alpha_{h,n}(s,a)g(u_n).\end{aligned}$$

Using a common argument from measure theory, this relation also holds for any integrable function by approximating with simple functions. The rest of the proof proceeds as in part a), with assumptions AI., AIV., and kernel conditions KI.-KIV. fulfilling the requirements of Hansen's Theorem 9. □

Definition. For a given MDP, let $Q(s_1,<a_1,a_2,...,a_n,t>)$ denote the expected discounted reward received when the process starts in state s_1 and actions $a_1,a_2,...,a_n$ are utilized consecutively

and the process then terminates. Formally

$$Q(s_1, < a_1, a_2, ..., a_n, t >) = E\left[\sum_{j=1}^{n} \gamma^{j-1} R(s_j, a_j)\right]$$

where the chance of transitioning to states $s_2, ..., s_n$ is understood to be governed by $P(s_j, \cdot, a_j)$.

The following lemma shows that state-action values possess a sort of continuity. This property will be needed to apply Lemma 7 to state-action values.

Lemma 6. *If assumptions AII., AIII., and AIV. hold, then for any n, and for any fixed but arbitrary sequence of actions $< a_1, a_2, ..., a_n >$, the function $f : S \to \mathbb{R}$ defined by*

$$f(s) = Q(s, < a_1, a_2, ..., a_n, t >)$$

is Lipschitz continuous.

Proof. Consider first the case $n = 1$. Let $s_1, s_2 \in S$. Then by AIII.

$$\begin{aligned} &|Q(s_1, < a_1, t >) - Q(s_2, < a_1, t >)| \\ =&|E[R(s_1, a_1)] - E[R(s_2, a_1)]| \leq C_r ||(s_1, a_1) - (s_2, a_1)||. \end{aligned}$$

Before considering the case $n > 1$, notice that by assumption AII.

$$\begin{aligned} |Q(s, < a_1, a_2, ..., a_n, t >)| &\leq E\left[\sum_{j=1}^{n} \gamma^{j-1} |R(s_j, a_j)|\right] \\ &\leq \sum_{j=1}^{\infty} \gamma^{j-1} C_0 \\ &= \frac{C_0}{1-\gamma}. \end{aligned}$$

Applying AIII. to the difference of means and AIV. to the difference of integrals, we have for $n > 1$

and $s_1, s_2 \in S$

$$\begin{aligned}
|f(s_1) - f(s_2)| =& |Q(s_1, < a_1, ..., a_n, t) - Q(s_2, < a_1, ..., a_n, t >)| \\
\leq& |E[R(s_1, a_1)] - E[R(s_2, a_1)]| \\
&+ \left| \gamma \int_S Q(u, < a_2, ...a_n, t >) P(s_1, du, a_1) \right. \\
&\left. - \gamma \int_S Q(u, < a_2, ...a_n, t >) P(s_2, du, a_1) \right| \\
\leq& C_r ||((s_1, a_1) - (s_2, a_1))|| + C_t ||Q(u, < a_2, ...a_n, t >)||_\infty ||(s_1, a_1) - (s_2, a_1)|| \\
\leq& C_r ||((s_1, a_1) - (s_2, a_1))|| + C_t \frac{C_0}{1-\gamma} ||(s_1, a_1) - (s_2, a_1)||.
\end{aligned}$$

By taking $C_1 = 2 \max\{C_r, \frac{C_t C_0}{1-\gamma}\}$ and inserting into the last inequality, we have

$$|f(s_1) - f(s_2)| \leq C_1 ||(s_1, a_1) - (s_2, a_1)||.$$

Finally, notice that $||(s_1, a_1) - (s_2, a_1)|| = ||s_1 - s_2||$ from properties of the Euclidean metric. We end with

$$|f(s_1) - f(s_2)| \leq C_1 ||s_1 - s_2||,$$

which proves the result. □

For the next lemma, suppose one has two MDPs defined on the same state and action space. For $(s, a) \in S \times A$, let $R^{(i)}(s, a)$ and $P^{(i)}(s, \cdot, a)$ denote the rewards and transition probabilities of process i for $i = 1, 2$. The lemma will show that if the expected rewards and transition probabilities of the processes are sufficiently close, then the expected discounted reward of a series of actions is also close.

Lemma 7. *Suppose the assumptions required for Lemma 6 hold. For any $\epsilon > 0$ and sequence of actions of length n, there exists $\delta_r(n, \epsilon)$ and $\delta_t(n, \epsilon)$ such that if $g : S \to \mathbb{R}$ is continuous and $||g||_\infty \leq \frac{C_0}{1-\gamma}$ and the following two conditions hold:*

1.

$$\sup_{(s,a) \in S \times A} \left| E[R^{(1)}(s, a)] - E[R^{(2)}(s, a)] \right| < \delta_r(n, \epsilon)$$

2.

$$\sup_{(s,a)\in S\times A}\left|\int_S g(u)P^{(1)}(s,du,a)-\int_S g(u)P^{(2)}(s,du,a)\right|<\delta_t(n,\epsilon),$$

then for any sequence of actions of length n,

$$|Q^{(1)}(s_1,<a_1,a_2,...,a_n,t>)-Q^{(2)}(s_1,<a_1,a_2,...,a_n,t>)|<\epsilon.$$

Proof. We proceed by induction on n, the number of actions to be executed. For $n=1$, take $\delta_r(1,\epsilon)=\epsilon$.

$$|Q^{(1)}(s_1,<a_1,t>)-Q^{(2)}(s_1,<a_1,t>)|=|E[R^{(1)}(s_1,a_1)]-E[R^{(2)}(s_1,a_1)]|<\delta_r=\epsilon.$$

Now assume the induction hypothesis holds for a sequence of actions of length $n-1$. Take $\delta_r(n,\epsilon)=\min\left\{\frac{\epsilon}{3},\delta_r(n-1,\frac{\epsilon}{3})\right\}$, and $\delta_t(n,\epsilon)=\min\left\{\frac{\epsilon}{3},\delta_t(n-1,\frac{\epsilon}{3})\right\}$. Then for a sequence of length n, we can condition on state s_2 and write

$$\begin{aligned}
&|Q^{(1)}(s_1,<a_1,...,a_n,t>)-Q^{(2)}(s_1,<a_1,...,a_n,t>)|\\
&\leq|E[R^{(1)}(s_1,a_1)]-E[R^{(2)}(s_1,a_1)]| &(2.6)\\
&+\gamma\left|\int_S Q^{(1)}(s_2,<a_2,...,a_n,t>)P^{(1)}(s_1,ds_2,a_1) &(2.7)\right.\\
&\left.-\int_S Q^{(2)}(s_2,<a_2,...,a_n,t>)P^{(2)}(s_1,ds_2,a_1)\right|. &(2.8)
\end{aligned}$$

Add and subtract $\int_S Q^{(1)}(s_2, < a_2, ..., a_n, t >) P^{(2)}(s_1, ds_2, a_1)$:

$$
\begin{aligned}
&|Q^{(1)}(s_1, < a_1, ..., a_n, t >) - Q^{(2)}(s_1, < a_1, ..., a_n, t >)| \\
\leq &|E[R^{(1)}(s_1, a_1)] - E[R^{(2)}(s_1, a_1)]| \\
&+ \left| \int_S Q^{(1)}(s_2, < a_2, ..., a_n, t >) P^{(1)}(s_1, ds_2, a_1) \right. \\
&\left. - \int_S Q^{(1)}(s_2, < a_2, ..., a_n, t >) P^{(2)}(s_1, ds_2, a_1) \right| \\
&+ \left| \int_S Q^{(1)}(s_2, < a_2, ..., a_n, t >) P^{(2)}(s_1, ds_2, a_1) \right. \\
&\left. - \int_S Q^{(2)}(s_2, < a_2, ..., a_n, t >) P^{(2)}(s_1, ds_2, a_1) \right|
\end{aligned}
$$

From Lemma 6 the value function is continuous, so by assumption the first difference can be made smaller than $\epsilon/3$ by choice of δ_r, the second difference made small by choice of δ_t, and the third difference made small by the induction hypothesis. Each difference is less than $\frac{\epsilon}{3}$, and the result is proved. □

2.4.3 Proof of Theorem

We are now prepared to prove Theorem 2.

Proof. The terms with subscripts will denote the values of the ARP with bandwidth h. The terms with no subscripts will refer to the original process. For any policy π, consider the difference

$$
\begin{aligned}
&|Q_h^\pi(< s, n >, a) - Q^\pi(s, a)| && (2.9) \\
\leq &|Q_h^\pi(< s, n >, a) - Q_h(< s, n >, < a, \pi(s_2), ..., \pi(s_m), t >)| && (2.10) \\
&+ |Q_h(< s, n >, < a, \pi(s_2), ..., \pi(s_m), t >) - Q(s, < a, \pi(s_2), ..., \pi(s_m), t >)| && (2.11) \\
&+ |Q(s, < a, \pi(s_2), ..., \pi(s_m), t >) - Q^\pi(s, a)| \,. && (2.12)
\end{aligned}
$$

By Lemma 4, m can be chosen large enough such that lines (2.10) and (2.12) are less than $\frac{\epsilon}{3}$. By Lemma 5, there exists an h and N such that if more than N iterations are run, the rewards and transition probabilities of the ARP become arbitrarily close to those of the real process. Furthermore,

by Lemma 3 there is a higher level N_1 such that if one starts at a level above N_1, the probability of straying below N after performing m actions is less than $\frac{(1-\gamma)\epsilon}{12C_0}$. When above level N, the requirements to apply Lemma 7 are met, and one can make line (2.11) less than $\frac{2C_0\epsilon}{12C_0-(1-\gamma)\epsilon}$. If the ARP does stray below level N, then the difference is bounded by $\frac{2C_0}{1-\gamma}$ as mentioned in Lemma 6. Then by conditioning on whether the ARP strays below level N,

$$\begin{aligned}&|Q_h(< s,n >,< a,\pi(s_2),...,\pi(s_m),t >) - Q(s,< a,\pi(s_2),...,\pi(s_m),t >)| \\ &\leq \frac{2C_0}{1-\gamma}\left(\frac{(1-\gamma)\epsilon}{12C_0}\right) + \left(\frac{2C_0\epsilon}{12C_0-(1-\gamma)\epsilon}\right)\left(\frac{12C_0-(1-\gamma)\epsilon}{12C_0}\right) = \frac{\epsilon}{3}.\end{aligned}$$

It follows that

$$|Q_h^{\pi}(< s,n >,a) - Q^{\pi}(s,a)| < \epsilon.$$

Because the policy π was arbitrary, we can replace it with a policy optimal for the real process, π^*. Then we know

$$\left|Q_h^{\pi^*}(< s,n >,a) - Q^*(s,a)\right| < \epsilon.$$

Now we claim that π^* also gives optimal values for the ARP. If it did not, and some other policy π_0 gave higher values, then consider that

$$|Q_h^{\pi_0}(< s,n >,a) - Q^{\pi_0}(s,a)|$$

can be made arbitrarily small, but this would imply $Q^{\pi_0}(s,a) > Q^{\pi^*}(s,a)$, which contradicts the choice of π^* as an optimal policy. We now have

$$|Q_h^*(< s,n >,a) - Q^*(s,a)| < \epsilon.$$

But by Lemma 1, $Q_h^*(< s,n >,a) = \widehat{Q}_h(s,a)$. Thus

$$\left|\widehat{Q}_{h,n}(s,a) - Q^*(s,a)\right| < \epsilon.$$

Since $(s,a) \in S \times A$ was arbitrary, the convergence is uniform. □

2.5 Experimental Results

Though the primary intent of this algorithm is to serve as an addition to the theory of continuous domain reinforcement learning, it is not impotent. With minor modification, it is capable of obtaining satisfactory solutions to standard benchmark problems in a reasonable amount of time.

In this section we detail an application to the Mountain Car problem [19]. The state space consists of two variables: the position p of the car, constrained to the interval $[-1, 1]$ and the velocity v, constrained to $[-3, 3]$. The action space consists of a single variable, a force applied to the vehicle along a line tangent to the road with a value in $[-4, 4]$. The car starts at the bottom of the hill at a standstill, corresponding to an initial state of $(-.5, 0)$. The goal is to drive the car to the top of the hill in the positive direction, i.e., $p > 1$, while keeping the velocity in bounds. If the goal is reached, a reward of 1 is experienced and the trial terminates. If the car goes out of bounds (that is, $p < -1$ or $|v| > 3$) then a reward of -1 is received and the trial terminates. In all other states the reward received is zero. The hill is defined by the expression

$$H(p) = \begin{cases} p^2 + p & \text{if } p < 0 \\ \frac{p}{\sqrt{1+5p^2}} & \text{if } p \geq 0 \ . \end{cases}$$

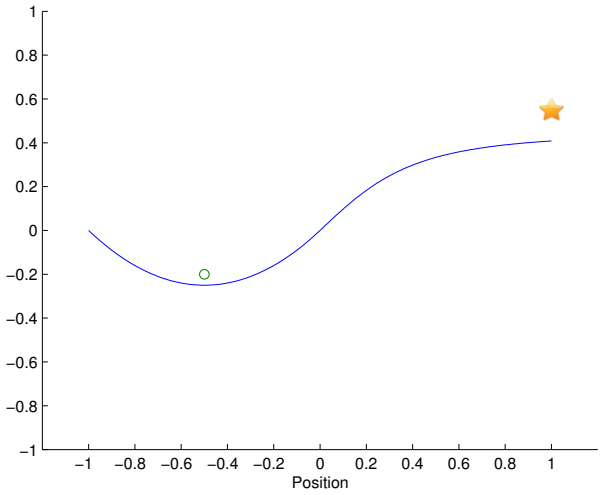

Figure 2.2: The hill for the mountain car to climb. The circle represents the initial position, and the star represents the goal.

This problem is non-trivial because the maximum force in the positive direction is not large

enough to move directly to the goal. Instead, the agent must learn to build momentum by alternating acceleration before trying to move to the goal.

2.5.1 Examination of Assumptions

Assumptions AI.-AIV., while sufficient to ensure convergence, are quite strong. In fact, there are many standard problems in Reinforcement Learning that do not meet them. For example, rewards are often of a discontinuous nature, where most states emit zero reward and crossing a boundary suddenly yields a reward or penalty. The purpose of this section is to show that the proposed algorithm is somewhat robust. That is, even in the presence of violated conditions, it can obtain policies adequate for solving the problem at hand. Specifically, we will see how the Mountain Car problem violates two out of four assumptions, yet the algorithm still produces a policy that can reach the goal state.

The first assumption is the continuous analogue of classic Q-learning's requirement that every state-action pair be visited an infinite number of times. We require the distribution of (s_n, a_n) to possess a density that is positive and sufficiently smooth. However, given the dynamics of the Mountain Car, there are areas of the state space that are inaccessible with the available actions. Consider the state consisting of position $p = .9$ and velocity $v = -2.9$. The position is at the top of the hill near the goal. The maximum acceleration in the negative direction is not strong enough to produce a negative velocity as large as -2.9 in the space given.

Figure 2.3 suggests that the density over the state space is adequately smooth, but it is clear that there are states which have not been visited. However, if the state space is restricted to the set of accessible states, it seems reasonable that the density is everywhere positive.

The second assumption requires bounded rewards. The Mountain Car problem clearly satisfies this.

The third assumption requires rewards to be Lipschitz continuous. It is common in Reinforcement Learning for rewards to be zero for most of the state-action space, with a goal state emitting positive rewards and penalties are issued for leaving boundaries or entering undesirable states. This is exactly the case for the Mountain Car. One could circumvent this problem by replacing impulse rewards with a low-variance Gaussian density function centered at the goal state, or use a similar method for smoothing rewards without disrupting the overall reward structure.

The fourth assumption requires weak convergence of transition probability measures as

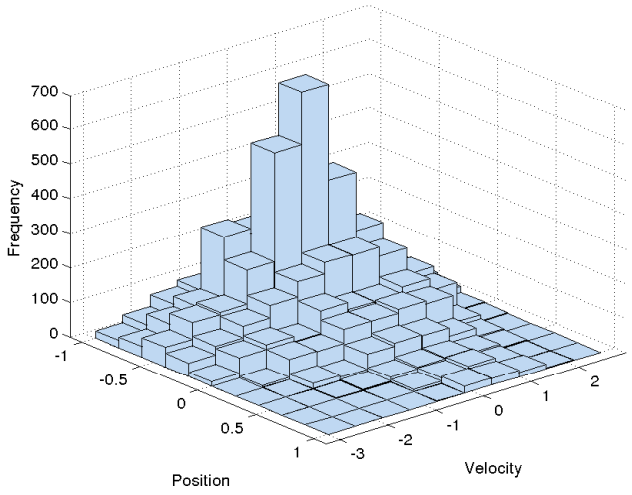

Figure 2.3: A histogram showing frequency of visits across the state space.

states and actions become more similar. Furthermore, that convergence must be uniform across the state-action space. For a problem with smooth, deterministic transitions such as Mountain Car, it is easy to verify convergence in distribution for u_n (the state being transitioned to) as state-actions (s_n, a_n) converge, which is equivalent to weak convergence. See Chapter 18 of Jacod & Protter [15].

It should be noted that the strength of the assumptions are directly linked to the desire for the value function estimate to converge uniformly. Kernel methods can achieve weaker forms of convergence with much weaker assumptions. For example, Devroye & Györfi [7] show that with proper bandwith selection, kernel density estimates can converge in $L1$ with no conditions on the density being estimated. This suggests that kernel-based algorithms may be able to return good policies despite discontinuities in the reward function, as is the case for the Mountain Car problem. However, this has not been thoroughly investigated for other problems yet.

2.5.2 Computational Considerations

An advantage of non-parametric approximators is that they have the potential to represent any function with arbitrary precision, but at a cost of increased computational load as more observations are used. The following paragraphs will discuss a few suggestions for alleviating this problem.

One idea is to be selective about which observations are kept for future calculations while

discarding the rest. For example, consider that before a boundary is reached for the first time, all rewards received are equal to zero, which is the initialized function value. Recording these observations will slow down future value function computations but does not add to the knowledge of the system. Suppose that when in state-action (s, a), reward r_n is received and the next state is u_n. One may choose to discard this experience if

$$|r_n + \gamma \sup_{b \in A} \widehat{Q}_{h,n-1}(u_n, b) - \widehat{Q}_{h,n-1}(s, a)| < \delta$$

for some tolerance δ. Only keeping experiences with the potential to improve the value function estimate ensures the algorithm learns efficiently.

Another idea is to, at some point, begin discarding old observations as new ones are recorded to keep a fixed number in memory. Consider that for terms $r_n + \gamma \sup_{b \in A} \widehat{Q}_{h,n-1}(u_n, b)$ calculated early, the value $\widehat{Q}_{h,n-1}(u_n, b)$ is not yet calculated with enough data to give a good estimate. Dropping old values in favor of new ones serves a double purpose of capping the increasing computational load and speeding convergence of value estimates. Care should be taken so that no region of the state-action space is left without observations for performing kernel regression.

A third option, and the approach taken in this example, is to "seed" the algorithm with a fixed number of randomly generated episodes that have reached the goal before beginning the standard ϵ-greedy exploration strategy. Exploration episodes are generated using random action selection. Episodes which reach a penalty state or fail to find the goal within 500 iterations are discarded, while successful episodes are passed to the algorithm. After 20 such episodes, the algorithm chooses random actions with probability ϵ and optimal (based on current knowledge) actions otherwise. At this point all iterations are passed to the learning algorithm.

As mentioned in the introduction, taking a supremum over all possible actions for a given state is, in general, a difficult problem. For the Mountain Car problem, which has only one action variable, a suitable choice for a kernel function allows for exact solutions. The Epanechnikov kernel is defined by

$$K^e(x) = \begin{cases} \frac{3}{4}(1 - x^2) & \text{if } |x| < 1 \\ 0 & \text{otherwise} \end{cases} \tag{2.13}$$

and is a common choice for kernel regression as it minimizes an approximation of mean integrated square error. See Section 3.3.2 of Silverman [25] for more details. For the three dimensional Mountain

Car problem, we define the multivariate kernel as the product of Epanechnikov kernels for each variable. That is, for state-action pairs represented by (p_k, v_k, a_k), we define

$$K((p_1, v_1, a_1) - (p_2, v_2, a_2)) = K^e(p_1 - p_2)K^e(v_1 - v_2)K^e(a_1 - a_2)$$

An advantage of formulating the kernel in this way is that the numerator of the derivative with respect to the action variable is quadratic in the action variable. Because the Epanechnikov kernel is non-zero on a finite interval, the domain of the action variable can be broken into a finite number of intervals on which the quadratic coefficients are constant. Thus all points which are candidate maxima are either a root of one of a finite number of quadratics or breakpoints between intervals. Using this method, the optimal action can be calculated relatively efficiently. If the routine finds multiple actions with the same optimal value, ties are broken randomly. Currently, this strategy works only on problems with one action variable, though the possibility of extending into higher dimensions is being investigated.

2.5.3 Results

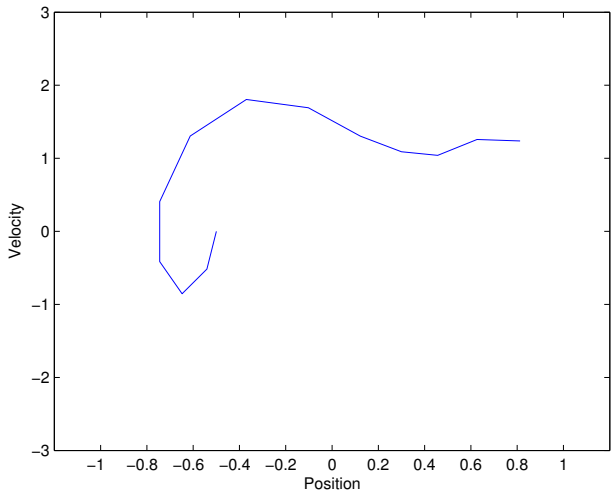

Figure 2.4: The trajectory through the state space as obtained by the final policy.

Implementation was in MATLAB 2012b in Ubuntu 12.04 on hardware with an Intel Xeon 3.47 gigahertz processor and 24 gigabytes of RAM. 7,500 iterations were recorded in 858 seconds. Parameter values were bandwidth $h = .2$, exploration parameter $\epsilon = .9$, discount factor $\gamma = .9$, and

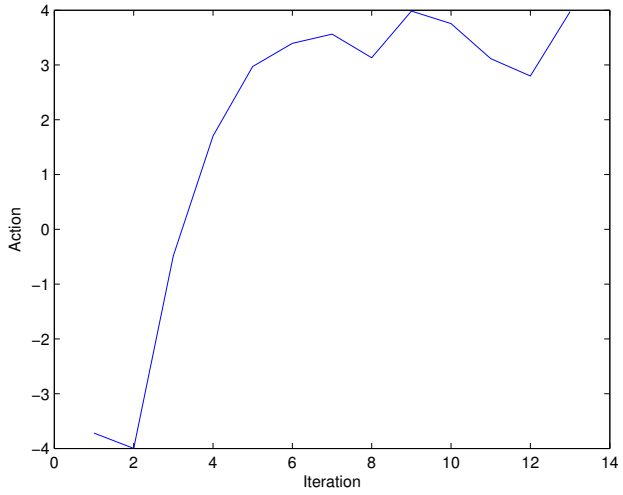

Figure 2.5: Actions chosen from the final policy.

$k = 20$ successful episodes to initialize with [1].

The policy obtained is near-optimal. Figure 2.5 shows action selection over time. The optimal policy will choose actions on the extreme ends of the action interval, and the learned policy sometimes chooses actions near but not at the boundary. Figure 2.6 shows iterations against time. There is an initially surprising phenomenon here. One would expect the number of iterations per second to increase as more data is collected, but after around 400 iterations the calculations actually speed up. The reason is that when only a small amount of data is available, the action-selection subroutine will find many actions with the same maximum value. Each best-action candidate is kept in memory until the tie is randomly broken at the end of the subroutine. As more data is collected, ties occur less commonly, and the subroutine finishes in less time. For a brief time, the increase in speed from faster action-selection outpaces the decrease in speed from accumulating data. Over time, this phenomenon vanishes, and the algorithm slows down once again. Figure 2.7 shows iterations against time for 25,000 iterations. From this plot it is clear that the described phenomenon is temporary.

Another unexpected occurence involved the optimal bandwidth. Repeating the experiment multiple times with different bandwidths revealed that the best results were obtained when the algorithm learned with a bandwidth of .2 but used a smaller bandwidth, .07, when building the final

[1]For the full source code, see the online appendix at `https://www.dropbox.com/s/8q9pd9tufc09tw5/DissertationOnlineAppendix.zip`.

policy. Early trials resulted in a policy that could find the goal but spent more time than necessary building momentum. Upon investigation, it was found that the value of state-actions corresponding to the initial state were calculated using values in the initial position but with positive velocity. This caused the policy to use positive acceleration (the best action when velocity is already positive) rather than negative acceleration (the best action when velocity is zero) from the initial state. A smaller bandwidth, specifically .07, avoided this issue. Note that using .07 when the algorithm was in the learning phase did not yield good results because it did not sufficiently generalize when a small amount of data was available. This suggests that even though the convergence proof uses a fixed bandwidth, in practice it is best to decrease the bandwidth as the amount of data increases.

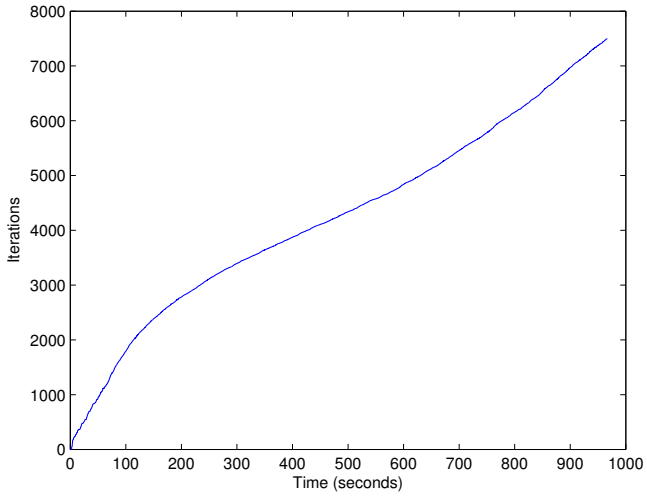

Figure 2.6: Graph of 7,500 iterations completed over time.

2.6 Discussion

This chapter has presented a reinforcement learning algorithm for continuous states and actions which is proven to converge to an optimal solution. Knowledge generalization is based on Nadaraya-Watson kernel regression. The most similar previous result is that of Ormoneit & Sen [21], which uses kernel-smoothing for problems with a continuous state space and a finite number of actions. The algorithm presented here is different in a few important ways. First, actions are allowed to be continuous. However, this introduces the burden of ensuring that small changes in actions result in small changes to transition dynamics. Second, the assumption on the distribution

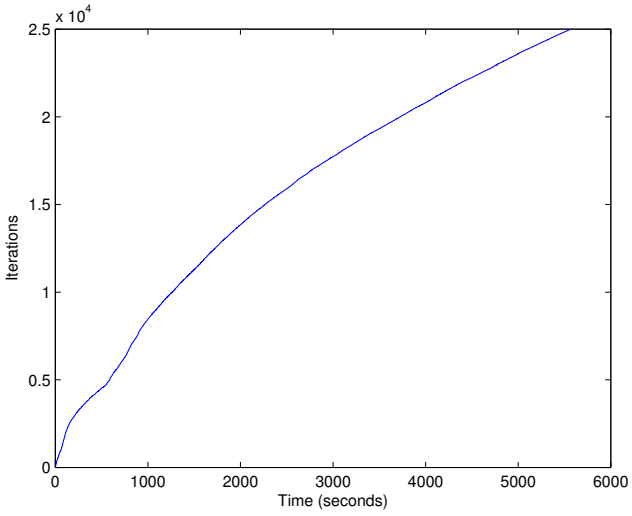

Figure 2.7: Graph of 25,000 iterations completed over time.

of states is weaker. Ormoneit & Sen's Assumption 3 requires states to be sampled uniformly from the state space. This has been weakened in the present paper to requiring the distribution of states to have a positive and smooth density.

One of the practical difficulties in implementing the algorithm is that the amount of computation required increases with each recorded observation. In addition to the ideas in Section 2.5, the author has experimented with sparsifying by selecting a grid of M points $\{(s_j, a_j)|j = 1, ..., M\}$ in the state-action space and assigning to each point a value such that performing kernel regression on those values will yield predictions similar to predictions obtained using kernel regression on the entire set of observations. Essentially, the idea is to use the kernel as a radial basis function at fixed locations in the state-action space and learn the weights for each. Specifically, suppose N transitions have been observed, $N \geq M$, with $\{(s_i, a_i)|i = 1, ...N\}$ and $\{y_{h,i} := r_i + \gamma \sup_{a \in A} \widehat{Q}_{h,i-1}(u_i, a)|i = 1, ..., N\}$ recorded. Define a matrix $X_{N \times M}$ entry-wise by

$$X_{i,j} = \frac{K_h((s_i, a_i) - (s_j, a_j))}{\sum_{j=1}^{M} K_h((s_i, a_i) - (s_j, a_j)}$$

and $\vec{Y}_{N \times 1}$ by $Y_i = y_{h,i}$. Solve for $\vec{\beta}$ that minimizes $||X\vec{\beta} - \vec{Y}||_2$. For a state-action (s, a), define $\vec{K}_{M \times 1}(s, a)$ by $K_j(s, a) = K((s, a) - (s_j, a_j))$. The value of (s, a) can then be estimated by $\vec{K}^T(s, a)\vec{\beta}$. For future observations, update $\vec{\beta}$ as described by Chambers [6]. This method requires computation that is linear in the number of observations and constant in storage requirements.

The fourth assumption concerns weak convergence of state transition probability measures. If the state-action space is separable, it is possible to define a metric on the set of all transition measures such that convergence in the metric is equivalent to weak convergence. See, for example, the Prokhorov metric [4]. Therefore it is possible to restate assumption four in a more succint manner that is sometimes easier to verify. However, extra work is then required to derive the properties needed in the convergence proof. The author has chosen to state assumption four in the manner that simplifies the proof.

Chapter 3

Regression Trees for Reinforcement Learning

The kernel-based algorithm of the previous chapter has excellent convergence properties, but computationally is a fundamentally expensive regression method and becomes more expensive as more data is accumulated. The goal of this chapter is to find a method for predicting state-action values that is more computationally efficient and better suited to higher dimensions and larger sets of transition data. Before introducing a new regression method, we will consider a different way of structuring the learning problem to take advantage of parallel computing.

3.1 Online and Offline Algorithms

Let us consider a Reinforcement Learning algorithm as consisting of two abstract tasks. The first task is data collection, and involves selecting actions (either randomly, choosing best known, or via some exploration method) and recording the data. The second task is learning, and uses the data obtained from the first task to do the actual updating of value estimates. We can classify algorithms according to how these tasks interact.

An *online* algorithm is one in which the tasks alternate. In all algorithms the data collection informs the learning, but in an online algorithm the learned value estimates can inform the data collection via influencing action selection. Intuitively, one can think of an online algorithm as

interacting with the system while learning. Both algorithms considered so far, standard Q-learning in Chapter 1 and the continuous variant in Chapter 2, are online algorithms. In fact, as presented, they are purely sequential online algorithms, as exactly one observation is made in between value estimate updates. The advantages of an online algorithm are the potential for smart exploration and the ability to be used in real time. The disadvantage is that learning is not as computationally efficient as offline algorithms. Figure 3.1 shows how online algorithms proceed.

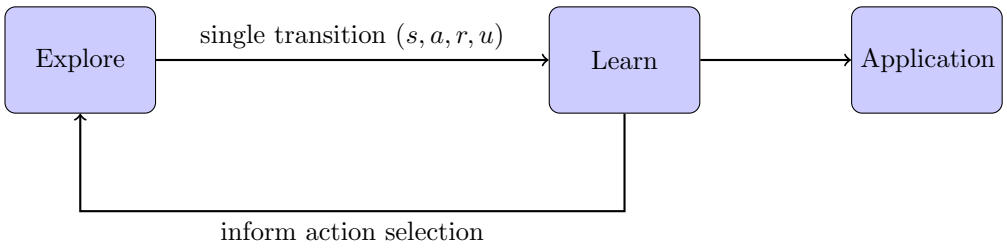

Figure 3.1: A sequential online algorithm interweaves exploration and learning. The exploration phase can use current knowledge to inform action selection.

An *offline* algorithm is one which separates the tasks of data collection and learning into distinct phases. Obviously, data collection occurs first. Because no learning takes place between iterations, the algorithm has no value estimates to guide action selection, so actions must be chosen randomly. Once the data collection task ends and all observations have been recorded, the learning phase begins. Because the algorithm now has access to an entire data set rather than a single observation, there are many options as to how to update value estimates. A common method is to use *batch* updates, which will be explained in more detail soon. The advantage of batch updates is that learning (and exploration, with a bit of work) no longer needs to be performed sequentially and can be completed in parallel, offering massive efficiency gains over online algorithms. The disadvantages are that offline algorithms are not capable of real-time application and that random action selection may not adequately explore the state-action space.

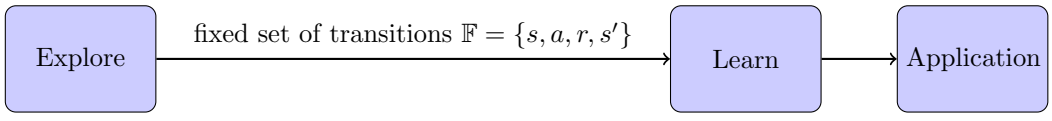

Figure 3.2: A batch-style offline algorithm separates completely the exploration and learning phases. Learning can be parallelized to be more efficient, but transition data may not adequately reveal system dynamics.

3.2 Fitted Q-iteration

The batch learning algorithm we will consider is *Fitted Q-iteration* [8]. This algorithm places no restrictions on how the data is collected, only that it is in the form of a collection of M four-tuples $\mathbb{F} = \{(s_k, a_k, r_k, u_k)|k = 1, ..., M\}$. Here s_k is the state at iteration k, a_k is the action, r_k is the reward, and u_k is the state to which the system transitions. Fitted Q-iteration also requires a regression method. Soon a specific regression method will be introduced, but in theory any method appropriate to the dimension of the state-action space method will work.

Initialize state-action value estimates $\widehat{Q}_0(s, a) = 0$ for all $(s, a) \in S \times A$. Then, for $n > 0$, repeat the following:

1. Build a training set $\mathcal{TS} = \{(i_k, o_k)|k = 1, \cdots, M\}$ of inputs and outputs where

$$i_k = (s_k, a_k) \tag{3.1}$$

$$o_k = r_k + \gamma \max_{a \in A} \widehat{Q}_{n-1}(u_k, a) \tag{3.2}$$

2. Use the regression algorithm to create $\widehat{Q}_n(s, a)$ from $\mathcal{TS}$

Continue until stopping conditions are reached. Notice that fitted Q-iteration reduces reinforcement learning to a sequence of regression problems where the response variable at each iteration is updated via the previous regression model and Bellman's equation. The explanatory variables are simply the state and action variables and do not change across iterations. Pseudocode for fitted Q-iteration is given by Algorithm 2.

Algorithm 2 Pseudocode for Fitted Q-iteration.

Input: $\mathbb{F}$, a set of M four-tuples; a regression method; a stopping condition; γ, a discount factor.
  Initialize $\widehat{Q}_0(s, a) = 0$.
  Initialize $n = 0$.
  for k=1:M **do**
    $i_k = (s_k, a_k)$.
  end for
  while Stopping condition not met **do** n = n + 1
    for k=1:M **do**
      $o_k = r_k + \gamma \max_{a \in A} \widehat{Q}_{n-1}(u_k, a)$.
    end for
    Build next regression model $\widehat{Q}_n(s, a)$ using $\{i_k|k = 1, \cdots, M\}$ as explanatory data and $\{o_k|k = 1, \cdots, M\}$ as response data.
  end while

Most of the work will be in building the regression model and finding a maximum of predicted values over action variables, so the choice of model has a dominant influence on computational speed. Kernel-based models may work well for problems like Mountain Car with 2-4 dimensions, but higher dimensional problems require a model better suited for large data and finding maxima over multiple action variables.

3.3 Regression Trees

One model better suited for higher dimensions is a *regression tree*. Rather than fit a global model, partition the input variable space and fit a local model on each partition. The partitioning is accomplished by a recursive series of binary splits, allowing the partitions to be represented visually by a tree graph, hence the name. The name *regression tree* is usually reserved for when all variables are quantiative while *classification tree* is used for qualitative variables. It is possible to construct trees that can deal with a mix of quantitative and qualitative variables, but for reinforcement learning problems we only need to deal with quantitative variables.

Construct the tree beginning with the root node which corresponds to all observations in the data set. Choose a "cut" variable and a "cut" value for that variable at which to split the data set. There are multiple specific methods for choosing the variable and value which will be discussed later. Create two child nodes from the root, a left child and a right child. All observations for which the cut variable is less than the cut value are sent to the left child, and all other observations are sent to the right child. Repeat this process for each child node, and then for all of their children nodes unless a node satisfies some stopping criterion. At that point, fit a local regression model to the observations belonging to that terminal node or "leaf." Most commonly the model is a constant prediction using the average response value for observations belonging to that node, but there is no reason a more complex local regression cannot be used.

The tree predicts a response value for a new observation by traveling down through the tree from the root node choosing the left child node when the observation's value for the cut variable is less than the cut value, and taking the right child otherwise. When a leaf is reached, the local regression model is used to produce a predicted value.

There are many variants on regression trees, with a few well known ones being CART [5], KD-Trees, Tree Bagging, Random Forests, CHAID, and QUEST. Below is a discussion of what

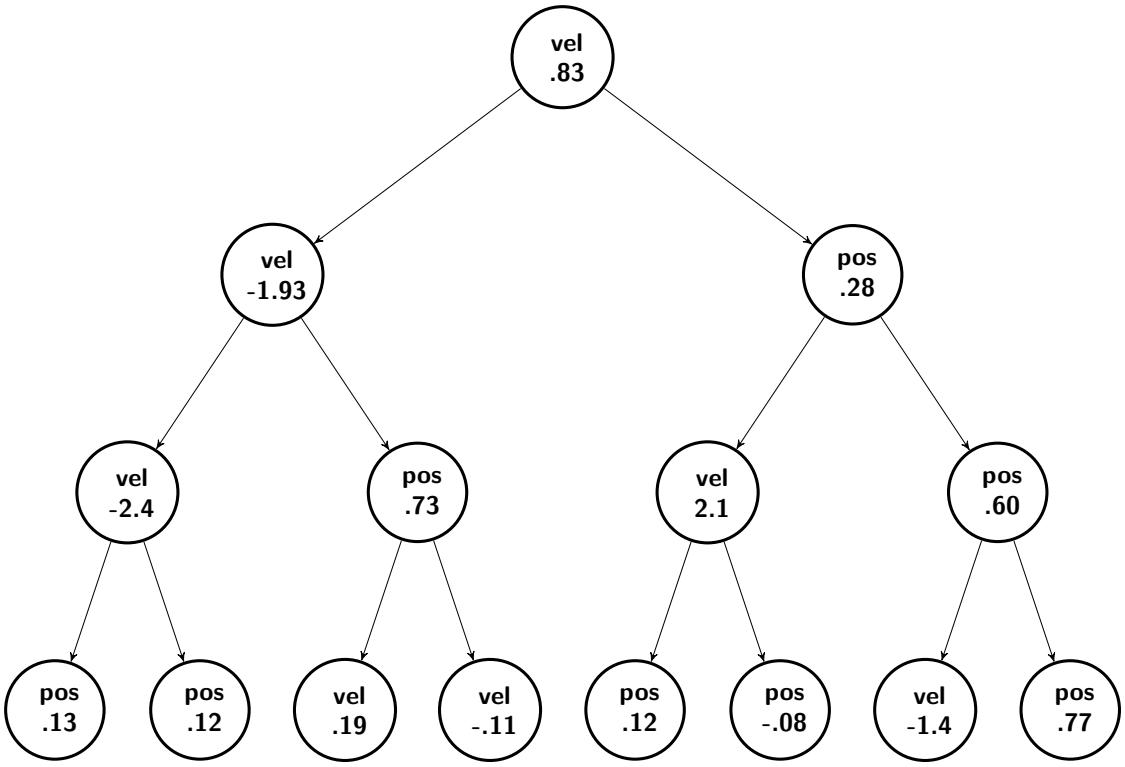

Figure 3.3: The first four levels of a regression tree approximating the value-function of the Mountain Car problem shown. Observations are sent left or right depending on whether the value of the listed variable is less than (sent left) or greater than (sent right) the listed value.

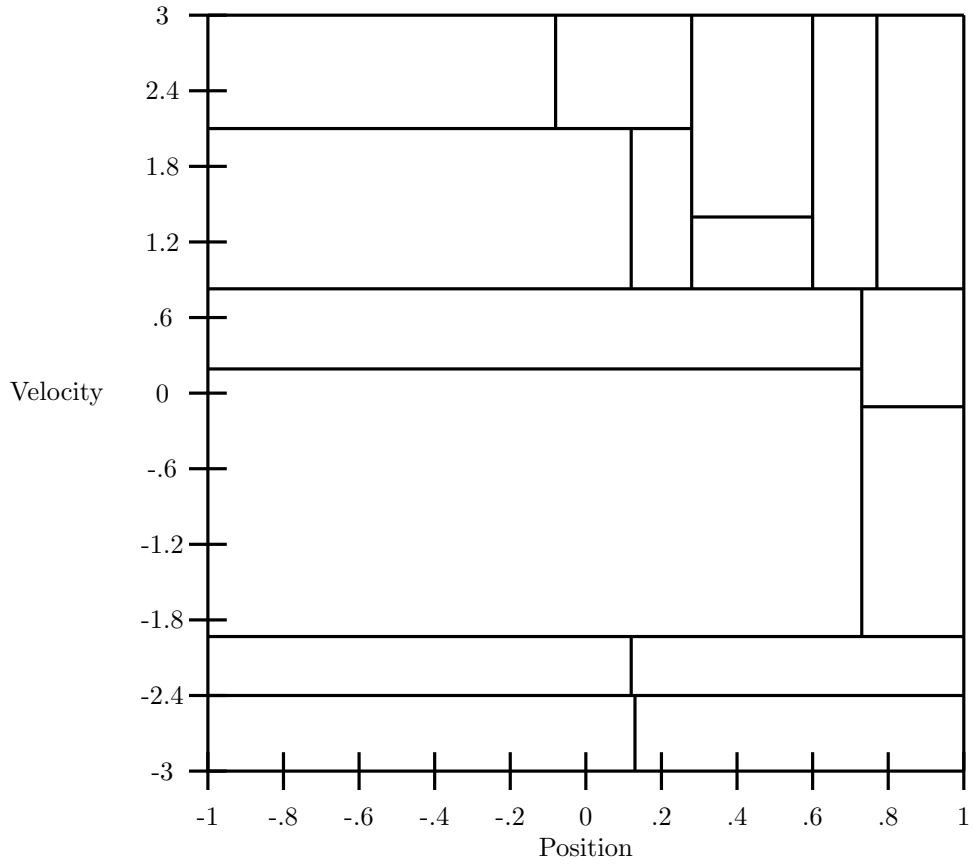

Figure 3.4: The state space as partitioned by the first four levels of the regression tree in Figure 3.3

properties may vary.

- **Number of trees** Some methods, like CART, build a single tree. Other methods build *forests*, multiple trees, and predict by taking an average of the predictions of each tree.

- **Split criteria** There are multiple ways of how to determine the cut variable and cut value. A few of the most common criteria are:

  - **MinSSE** Pick the variable and value such that the sum squared error across both of the resultant children nodes is minimized. Specifically, pick the cut so that

$$\sum_{k \in leftnode} (y_k - \bar{y}_{leftnode})^2 + \sum_{k \in rightnode} (y_k - \bar{y}_{rightnode})^2$$

    is minimized. This is a relatively expensive way of selecting cuts, but observations in a leaf have homogeneous response values.

  - **KD-Median** Choose the cut variable by making an ordered list of variables and simply cycling through them, choosing the one least recently split by. Choose the cut value by taking the median of values of the cut variable for observations belonging to the node. This builds trees very quickly, but response value in a leaf are probably less homogeneous than MinSSE.

  - **Random** Choose the cut variable and value uniformly and randomly. This is typically used with forests where multiple trees strengthen the weak predictive power of any particular tree.

- **Stop splitting criteria** When should a node be split into children, and when should it be left as a leaf?

  - **MinParent** Do not split a node unless it has a minimum number of observations. The specific value is a parameter input by the user or set as a percentage of the total number of observations.

  - **MinLeaf** Do not split a node unless each child would have a minimum number of observations. The practical difference between MinParent and MinLeaf criteria is subtle, but suppose MinSSE was used and selected 1 observation for the left node and 9 observations

for the right child. Using MinChild can prevent overfitting by disallowing leaf nodes with a small number of observations.

  - **Purity** Do not split a node with response values that are already homogeneous enough. Given a purity tolerance parameter δ, do not split a node if

$$\frac{\sum_{k \in node}(y_k - \bar{y}_{node})^2}{n} < \delta$$

    where n is the number of observations corresponding to the node.

- **Pruning** Pruning prevents overfitting and reduces complexity by removing portions of the tree which do not add to predictive accuracy. For example, a pruning algorithm may use cross-validation to make predictions with and without a portion of the tree. If that portion does not aid prediction, it is removed. Many tree variants purposely initially overfit and rely on pruning to obtain a quality tree.

- **Local prediction method** Most tree variants simply fit a constant prediction model at each leaf as the average value of the repsonse variable for observations belonging to that node. It is also possible to fit linear, kernel, or some other model and let the tree serve simply as a fast device for clustering the data.

Regression trees have been used in the context of fitted Q-iteration for reinforcement learning problems with continuous states and finite actions. In Ernst et al. [8] the authors grow a separate tree (or forest) for each distinct action, building the tree by only splitting state variables. Finding a maximum over actions for a given state is accomplished by using each tree to make a prediction for that state, then picking the action with the highest predicted value.

The theme of this dissertation is continuous action reinforcement learning, and growing a tree for each of an uncountable number of actions is clearly not feasible. The remainder of this chapter will detail modifications for using regression trees with continuous actions.

3.3.1 Regression Trees for Continuous Action Reinforcement Learning

A regression tree for continuous action learning requires additional functionality in two ways. The first is found in building the outputs for the training set:

$$o_k = r_k + \gamma \max_{a \in A} \widehat{Q}_{n-1}(u_k, a).$$

Specifically, consider the term $\max_{a \in A} \widehat{Q}_{n-1}(u_k, a)$. We need to be able to find a maximum predicted value over the action variables. Secondly, when the batch algorithm is complete, a final policy is built by setting

$$\pi^*(s) = \operatorname*{argsup}_{a \in A} Q^*(s, a).$$

We need not just the best predicted Q-value for a given state, but the particular action variable values that produce that value. A leaf of the tree will correspond not to one particular value for the action variables, but to an uncountable subset of values for the action variables. How should specific action variable values be picked?

Therefore, in addition to the properties described in the previous section, a continuous action tree must also decide how to implment additional functionality.

- **Whether to split action variables**
  - **RegTree** The most straightforward approach is to build a single tree over the state-action space allowing nodes to split at state or action variables. This approach keeps the standard structure of the tree, simply supplying a sub-algorithm to perform the optimization. This approach will be referred to as a *RegTree*, where the "reg" doubly refers to regression and regular, as it is closer to regular tree structures compared to the other modification to be presented.
  - **RLTree** Notice that in the context of reinforcement learning, the only values that are predicted are maximums over action variables. Response variable values for non-optimal actions are irrelevant. Therefore, one could build a tree by only splitting state variables, then maximizing over the action variables without splitting them. This approach is only suitable for reinforcement learning purposes, thus it will be referred to as the *RLTree* approach.

- **Action selection** Once a node with maximum response value has been found, how should a representative action be chosen?
  - **NodeMiddle** Each node represents a hyperrectangle in the state-action space. For each action variable, simply take the average of the bounds defined by the rectangle. This is a simple and fast way to select actions, effectively discretizing actions according to the structure of the tree and the amount of data. Notice this will work only with the RegTree approach. The RLTree approach, which does not split nodes by actions, will always give the same action value (the middle value for each action variable).
  - **BestObserved** If the MDP has deterministic or low variance system dynamics, then select the action with the highest observed value. Look at all observations corresponding to the node, find the one with the highest $\widehat{Q}_n(s,a)$, and use that action. If system dynamics are not near-deterministic, this may capture noise rather than average values. When appropriate to use, it works equally well with RegTree and RLTree approaches.
  - **LocalOptimization** If the tree fits a local prediction model more complex than the average of response values, one can try to find the action that maximizes the prediction response. For a linear prediction model with no coefficients equal to zero, this amounts to finding the corner of the hypercube with the best value. For a kernel or more complex model, there may not be a straightforward way to find the best action, and a general purpose optimizer may be used at high computational cost.
- **When to find best action** Unless a tree is grown as completely as possible with a leaf for each observation, there will be more observations than leaf nodes. Therefore during Fitted Q-iteration when the output is calculated for each state-action observation, each leaf node will be visited multiple times. Hence the action selection procedure will repeat itself needlessly.
  - **PreCalculated** Significant efficiency gains can be achieved by calculating and storing the best value and best action for each leaf node when the tree is constructed. This increases slightly the time to construct the tree, but is more than made up for by time saved in Fitted Q-iteration output calculation.
  - **OnDemand** There are special cases in which finding actions during Fitted Q-iteration is preferred to pre-calculating and storing. Notice that if constant value prediction is used,

the exact state variable values are, beyond the fact that they led to this node, irrelevant. However, if linear, kernel, or a more complex prediction method is used, the exact state variable values can be used along with the best action variable values to calculate $\widehat{Q}_n(s,a)$ with greater precision. Therefore, finding the best actions on demand is useful when value precision is preferred over quick computation.

3.4 Implementation

At this point, it is becoming clear that there are an overwhelming number of ways of building a tree. This section will describe the choices made for building a flexible yet manageable regression tree implementation for continuous action reinforcement learning.

- **Single tree** We will focus on building a single strong predictor rather than a forest of weak predictors.
- **MinSSE, KD-Median** We will allow cuts based on minimizing SSE across children nodes or by the median value of the variable split by least recently. Note that any particular tree will only use one of these methods at a time, but the choice is available before the tree is grown. We will not allow random cuts as this is only useful when building forests.
- **MinParent, MinLeaf, Purity** These criteria for stopping splits are not mutually exclusive, so we will use all three for each tree built.
- **No Pruning** Wise choice of MinParent, MinLeaf, and Purity parameters will prevent overfitting sufficiently, so pruning is redundant and adds unnecessary complexity to the code.
- **Constant, Linear, Kernel** The user can pick whichever local prediction method they prefer, though only one at a time can be used in a particular tree.
- **RegTree, RLTree** The user can pick whether to split over action variables (RegTree) or not (RLTree). Only one method can be used on a particular tree.
- **NodeMiddle, BestObserved, LocalOpt** If a RegTree tree uses constant prediction at a leaf, the user may choose to select actions by taking the middle of the node or the action with the best observed value. If a RLTree tree uses constant prediction, choosing the action with

the best observed value is the only sensible choice. If a tree of either variety uses linear or kernel prediction at a leaf, the action will be selected by local optimization.

- **PreCalculated, OnDemand** If constant prediction is used at a leaf, there is no reason not to precaculate the best action and value for that action during tree construction. If linear or kernel prediction is used, it is assumed the user desires precision over speed, thus actions are selected on demand during Fitted Q-iteration.

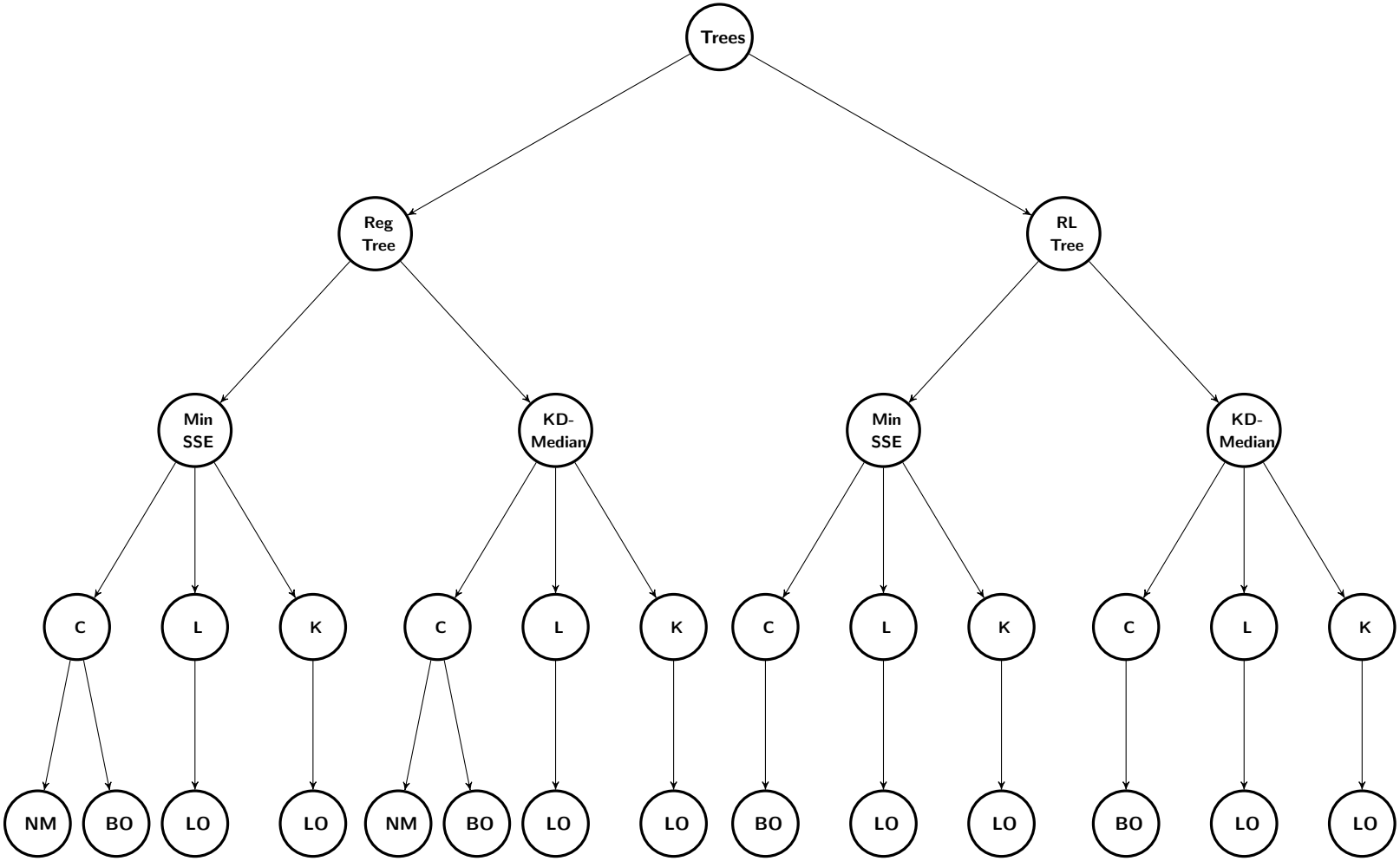

Figure 3.5: A visual display of all possible tree variants. The third level represents the local prediction method used where C stands for constant, L stands for Linear, and K stands for Kernel. The fourth level represents the method for selecting actions where NM stands for NodeMiddle, BO stands for BestObserved, and LO stands for LocalOptimization.

Figure 3.5 classifies the tree variants by using, appropriately, a tree format. [1]

3.5 Experiments and Comparison

First, in order to compare against the results of Chapter 2, we will examine a policy obtained by a tree variant on the Mountain Car problem. Then, each tree variant will be run under similar parameters in order to compare their computational efficiency. Then, for the sake of variety, the Inverted Pendulum problem will be introduced, and a solution will be examined.

For both experimental problems in this chapter, we will generate transition data quasi-randomly, using a 3-dimensional Halton sequence. See Appendix A for details on constructing the sequence.

3.5.1 The Mountain Car Problem

We will consider a solution to the Mountain Car problem obtained from a RegTree (splits over action variables allowed) using the MinSSE method to determine splits, and choosing actions using the NodeMiddle method. This tree variant is used for comparison because these choices are, in the author's opinion, either the most natural or the most commonly used. Specific values for the tree parameters are in Table 3.1.

Parameter	Value
Type of tree	RegTree
Split criteria	MinSSE
Action selection	NodeMiddle
Transitions	20,000
Iterations	20
MinChild	5
MinParent	10
Purity	10^{-7}
Discount factor γ	.8

Table 3.1: The specific methods and values used in the tree variant for solving the Mountain Car problem.

Using a 6-core processor running Fitted Q-iteration in parallel, the algorithm completed in 36.82 seconds. Compare to the 858 seconds for the algorithm in Chapter 2 to complete. Incidentally,

[1] See the online Appendix at `https://www.dropbox.com/s/8q9pd9tufc09tw5/DissertationOnlineAppendix.zip` for MATLAB code for each of the variants.

using less data and fewer iterations, the algorithm was able to produce less-optimal policies that were still capable of reaching the goal in as few as 13 seconds. Figure 3.6 shows the final policy obtained.

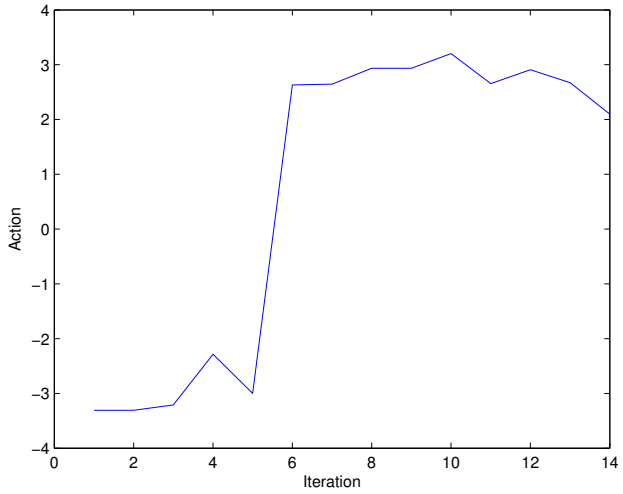

Figure 3.6: The policy obtained using Fitted Q-iteration and a regression tree.

Qualitatively, the policy is quite similar to Figure 2.5 from the previous chapter. Do notice that the actions, bounded by -4 and 4, are not as close to their boundaries in the tree variant. This is due to the NodeMiddle method of selecting representative actions. Other methods, such as maximizing a linear model, can better select actions near the boundaries, but at increased computational cost.

3.5.1.1 Comparing Solution Time

To compare the computation time for tree variants, we will solve the problem with the same number of observations and iterations while keeping as many tree parameters as possible constant. Table 3.2 gives the values for those parameters, and Table 3.3 gives the amount of time needed to complete all iterations with the given number of transitions.

Some of the results are what one would expect intuitively, but some are not. A first observation is that the local prediction method has far and away the greatest impact on computation time. Kernel regression takes around 25 times as long to complete as a constant prediction method, and linear regression takes around 50 times as long. The increase in time for kernel regression is perhaps

Parameter	Value
Transitions	20,000
Iterations	20
MinChild	5
MinParent	10
Purity	10^{-7}
Discount factor γ	.8
Kernel Bandwidth	.1

Table 3.2: The tree parameters kept constant across all tree variants for solving the Mountain Car problem.

Type of Tree	Split Criteria	Regression/Action Selection	Time (seconds)
RegTree	MinSSE	Constant/NodeMiddle	36.82
		Constant/BestObserved	22.22
		Linear	1491.18
		Kernel	659.45
	KD-Median	Constant/NodeMiddle	18.66
		Constant/BestObserved	23.76
		Linear	2087.58
		Kernel	1156.72
RLTree	MinSSE	Constant/BestObserved	22.44
		Linear	1367.67
		Kernel	536.83
	KD-Median	Constant/BestObserved	20.44
		Linear	1736.95
		Kernel	113.39

Table 3.3: Time needed for each tree variant to process 20 iterations at 20,000 transitions.

not as surprising as that for linear. This seems due to using MATLAB's built-in linear regression tools. However, even if a custom-coded linear solver is used, constant prediction is much too fast while still being accurate for either linear or kernel prediction to compete.

A second observation is that the difference between RLTrees and RegTrees is not nearly as dramatic, but RLTrees seem to have a slight advantage, completing in about 80-90% of the time for the corresponding RegTree to complete. An outlier to this rule does exist: compare the times for KD-Median/Kernel trees for RLTree and RegTree. The RLTree takes roughly 10% as much time to complete. The most likely reason for this huge difference is how the subroutine for maximizing kernel predictions over actions deals with multiple actions with the same maximum value. All candidate maximum actions are kept in memory until the tie is broken randomly at the end, and manipulating and storing the arrays containing the candidate maximum actions increases

computation time. Depending on the particular structure of the function being maximized over, there may be very many ties or there may be very few, allowing for significant variation in the completion time for tree variants with kernel prediction.

A third observation is that while one would expect KD-Median to complete in less time than MinSSE since the split criteria is much simpler, this is sometimes true but not always. For example, consider the RegTree/NodeMiddle trees for KD-Median and MinSSE. KD-Median takes half the time to complete, which agrees with intuition. However, for the RegTree/Linear trees, MinSSE completes faster. This seeming paradox is resovled when one considers the difference in computation time for *building* the tree and using the tree to *predict* state-action values. In a tree such as RegTree/NodeMiddle where prediction is very fast, the time it takes to build the tree dominates. Since KD-Median does indeed build a tree faster than MinSSE, we see a shorter computation time for KD-Median. On the other hand, in a RegTree/Linear tree, the prediction task is more intensive than the building task. Since KD-Median cyles through variables, the single action variable is split just as often as each of the state variables. Meanwhile MinSSE, with its more complex split criteria, finds greater value homogeneity when action variables are split less often than state variables, leading to a greater number of paths to search down the tree when seeking optimal actions. The greater number of paths means prediction must be performed at a greater number of terminal nodes. Therefore, since the prediction task dominates and must be performed more often, KD-Median non-intuitively leads to an increase in total computation time.

3.5.2 The Inverted Pendulum Problem

Consider a cart on a horizontal track with a massless rod attached to the top via a hinge, so that the rod may rotate through the half-circle above the top of the cart. The other end of the rod is attached to a spherical mass, hence the Inverted Pendulum. The goal is to balance the pendulum above the cart, but the system is obviously inherently unstable. An agent controls the system by imparting a force on the cart parallel to the horizontal track. Figure 3.7 shows the problem setup. We will attempt to find a solution when the mass of the cart is 1 kilogram, the length of the rod is 1 meter, and the mass of the pendulum is .2 kilograms.

The state space is defined by 2 variables: θ, the angular deviation of the rod from vertical, and θ_{vel}, the angular velocity of the rod. θ is allowed to be in the interval $[-\frac{\pi}{4}, \frac{\pi}{4}]$ and θ_{vel} is allowed to be in the interval $[-\pi, \pi]$. If either variable leaves its respective interval, the system is considered

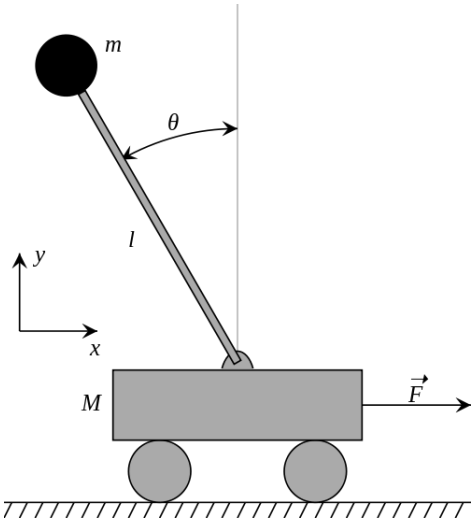

Figure 3.7: A diagram of the Inverted Pendulum problem.

to be in a penalty state, a reward of -1 is assessed, and a new trial starts. If the system is not in a penalty state, the reward is zero. The action space is defined by a single variable, the force imparted to the cart in Newtons. The force is constrained to the interval [-10, 10].

The dimensionality of the problem is the same as the Mountain Car problem, so the computational complexity is similar but there are important differences. First, the Mountain Car problem has a "bang-bang" optimal policy, so it tests a regression model's ability to find maxima near boundaries. An optimal policy for the Inverted Pendulum problem will actually avoid extreme actions, so this problem will test an algorithm's ability to use a fine touch when choosing actions. Secondly, the Mountain Car is a goal-state driven problem. Exploration must be thorough enough to find the goal state, but only a small region of the state-space is actually visited in an optimal trajectory, as can be seen in Figure 2.4. The Inverted Pendulum has no goal state, and is instead driven by the penalties assessed when the pendulum falls over or reaches a high angular velocity. Thorough exploration is more important here, as the instability of the pendulum can lead a policy into a large region of the state space. A good policy must be able to respond to a wide variety of states.

We will use the same tree variant that was used to solve the Mountain Car problem to solve this problem. Table 3.4 shows the parameters used.

A solution was reached in 168.37 seconds. While testing a policy for the Mountain Car problem is straightforward (simply begin from the starting position and see if the goal is reached),

Parameter	Value
Type of tree	RegTree
Split criteria	MinSSE
Action selection	NodeMiddle
Transitions	100,000
Iterations	30
MinChild	5
MinParent	10
Purity	10^{-7}
Discount factor γ	.9

Table 3.4: The specific methods and values used in the tree variant for solving the Inverted Pendulum problem.

we desire the pendulum to remain upright under a variety of conditions. Therefore, testing an Inverted Pendulum policy will consist of a series of 10 trials of 100 time steps each. For each trial, the system initializes with a random angular position drawn uniformly from $[-\frac{\pi}{8}, \frac{\pi}{8}]$ and a random angular velocity drawn uniformly from $[-\frac{\pi}{2}, \frac{\pi}{2}]$. The system must remain in bounds for the full duration of each trial to be considered a success. Figure 3.8 shows the trajectory for each of ten trials.

Notice that the policy produces two stable points, to the left and right of vertical, where the force from acceleration balances gravity and produces zero angular velocity. The angular position stays within a small region in a dynamic equilibrium, keeping the pendulum from falling indefinitely.

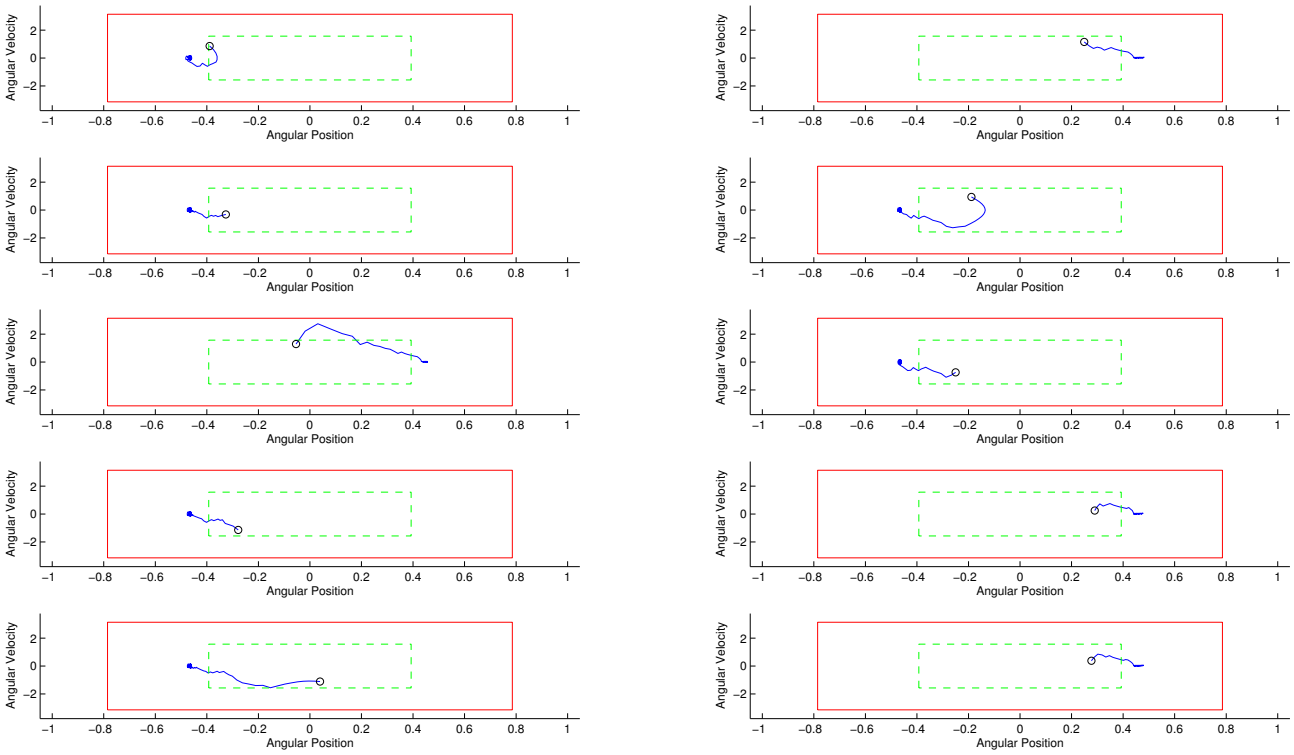

Figure 3.8: The trajectories for ten trials. States outside the red solid box are penalties. Trials begin randomly in the green dashed box. The black circle marks the actual starting state for the trial.

Chapter 4

Artificial Penalty Learning

In Chapter 2, the algorithm was online and sequential. Data was collected by interacting with the system, choosing actions based on current value estimates. Exploration of the state-action space was, in a sense, "smart" because it was guided by what was already learned.

In Chapter 3, the algorithm was offline and batch. Figure 4.1 reminds the reader of the complete separation between the exploring and learning tasks for a batch algorithm. Since a generative model for the Mountain Car and Inverted Pendulum problems was available, experiments used quasi-random sequences to generate transition data that uniformly represented the state-action space.

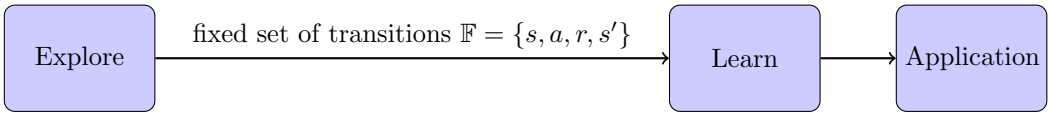

Figure 4.1: A batch-style offline algorithm separates completely the exploration and learning phases. If a transition-generating model is not assumed, exploration takes the form of trajectories driven by random action selection.

However, it is typically assumed that a generative model is not available. Data must be collected from trajectories using random actions. This may result in a set of transitions that does not represent important regions of the state-action space, such as states near a goal. Clearly, this can have a significant effect on the quality of the final policy produced.

Consider a goal-driven problem in which the agent is in an environment fraught with states from which the agent can no longer succeed and must reset to an initial state. Examples include

navigating a field with pits into which the agent can fall, steering a bicycle, helicopter, or any device which is capable of crashing, or traversing a hilly region with valleys that cannot be escaped. Such a problem will have a very low probability for a randomly exploring trajectory to reach the goal, requiring a massive number of trajectories before the number of transitions reaching the goal is sufficient for the learning phase. This has a two-fold negative effect. First of all, it will take time to collect or generate enough trajectory data. Also, the learning phase will take a significant amount of time to process the huge amount of data.

4.1 Artificial Penalty Learning

This chapter proposes a limited form of learning which we call Artificial Penalty Learning (APL) for the exploration phase. This type of learning yields more thorough exploration of the state-action space in fewer transitions and has an increasing chance of reaching the goal for each consecutive trajectory. An example problem will demonstrate superior performance over pure batch.

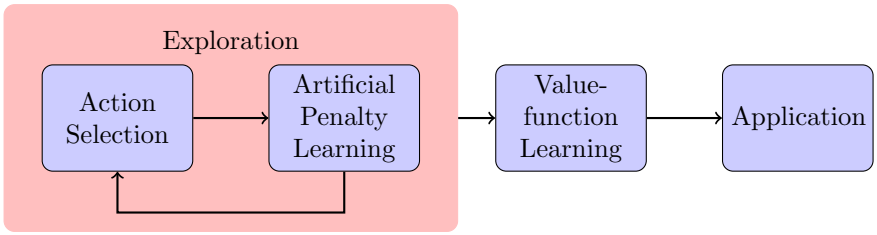

Figure 4.2: In Artificial Penalty Learning, the exploration and value-function learning phases are still separate, but the exploration phase uses a limited, efficient style of learning to improve the data passed to the value-function learning phase.

First, we will consider a state a *natural penalty* if it is explicitly defined by the dynamics of the Markov Decision Process to represent failure and force the system back to an initial state. Usually this is accompanied by a negative reward, so that the learning phase will see this as an undesirable state. We have seen examples of natural penalties in the Mountain Car and Inverted Pendulum problems. For example, if the angular position of the pendulum falls outside the interval $[-\frac{\pi}{4}, \frac{\pi}{4}]$, a reward of -1 is received and a new trial begins.

Now, consider that there also exist states which are not dictated by the MDP to be natural penalties, but will lead to a natural penalty no matter what action is used. As an example, if the state of the pendulum is $\theta = \frac{\pi}{4} - .04$ and $\theta_{vel} = \frac{e\pi}{4}$, the pendulum is leaning too far for any allowable

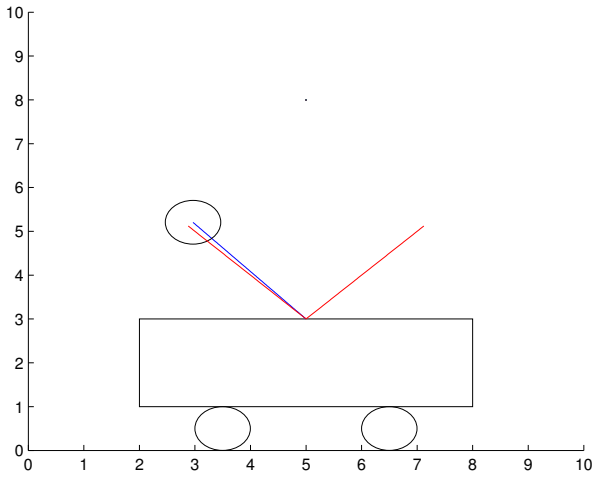

Figure 4.3: The Inverted Pendulum problem when $\theta_{pos} = \frac{\pi}{4} - .04$ and $\theta_{vel} = \frac{3\pi}{4}$. The maximum allowable force to the left will not prevent the angular position from crossing the red penalty boundary.

action to overcome the force of gravity. At the next time-step, the angular position will be greater than $\frac{\pi}{4}$. To borrow a phrase from chess, "checkmate in one." Though the MDP does not define this state as a penalty, the agent has already lost control and for all practical purposes is in a penalty.

Define an *artificial penalty* to be a state which is not a natural penalty, but all actions from this state lead to penalties. Once enough artificial penalty states have been recognized, there may be states for which all actions will lead to either a natural or artificial penalty. These states will be classified as artifical penalties themselves. Proceeding in this manner, portions of the state space will be classified as undesirable and unworthy of further exploration. For the remainder of this chapter, the term "penalty" will refer to both natural and artificial penalties. The distinction will be made when important.

APL breaks down into three tasks:

1. Build an efficient data structure for recording whether a state-action leads to a penalty.
2. Recognize when all actions for a particular state lead to a penalty, and classify that state as a penalty.
3. Select actions randomly and uniformly from those which have not been observed to lead to penalties.

Depending on the specific dynamics of the problem, there is potential for the non-penalty state space to be significantly reduced. If the agent only considers actions which have not been observed to lead to penalties (natural or artificial) from the current state, the effect is exploration that is concentrated in areas more relevant to good policies.

For problems with continuous states and actions, we will work with a discretization of the state-action space. For APL to be effective, we do not want states which are part of optimal trajectories to become penalties. Thus we need to assume one of the following holds:

- State transitions are continuous in the sense that similar actions used in similar states will lead to similar states.
- An optimal trajectory will not be too close to penalties.

Algorithm 3 gives pseudocode for APL.

Algorithm 3 Pseudocode for Artificial Penalty Learning.

Input: An MDP with discretized state-action space, number of transitions to record M
  Initialize state s_0
  for n=0:M **do**
    Choose a_n randomly and uniformly such that
    (s_n, a_n) has not been marked as leading to a penalty
    Receive next state u_n and reward r_n.
    Check if u_n is a penalty state.
    if u_n is a penalty state **then**
      Mark (s_n, a_n) as leading to a penalty.
      if All actions for s_n lead to a penalty **then**
        Mark s_n as a penalty.
      end if
      Set s_{n+1} as an initial state
    else
      Set $s_{n+1} = u_n$
    end if
  end for

4.2 Penalty Trees

The goal for APL is to produce good policies in as little time as possible. Therefore, the exact method used to discretize the state-action space is of key importance, as the three tasks of APL must be completed as efficiently as possible. Given the success of tree-based binary splits in the last chapter, we will use a tree as the data structure for discretizing the state-action space.

A *penalty tree* is a fully grown binary tree. Unlike a regression tree, which partitions the space unevenly based on supplied data, a penalty tree partitions the space into identical hyper-rectangles. All nodes at a particular level of the tree split at the same variable. The user inputs the number of times each variable is to be split. First, the state variables are cycled through until each is split the appropriate number of times. Nodes at this level are important because they represent portions of the state space. Then, action variables are cycled through until each is split the appropriate number of times. The upper part of the tree partitions the state space, and the bottom part of the tree partitions the action space.

A terminal node represents a small portion of the state-action space, but rather than assigning the leaf a value corresponding to a predicted Q-value, the leaf is assigned a binary variable *penalty* which takes value 0 if that state-action has not been observed to lead to a penalty, and value 1 if it has. For parent nodes of leaves, the variable *penalty* is real-valued and is the average of the *penalty* variable for its children. Traveling back up the tree, the value of *penalty* for every non-terminal node is calculated as the average of *penalty* for its two immediate children node. At the first level at which an action variable splits (the nodes of which represent partitions of the state space), the value of *penalty* is interpreted as the percent of actions which have been observed to lead to a penalty state. If $penalty = 1$ for one of these nodes, then the corresponding state is classified as an artificial penalty.

The *penalty* variable also drives random action selection. Given a particular state, travel down the tree until nodes begin splitting at actions. Suppose this results in arriving at node n. Let p_l and p_r denote the values of the penalty variable for the left and right children of node n. With probability

$$\frac{1 - p_l}{2 - p_l - p_r}$$

travel to the left child of n; otherwise travel to the right child. Repeat this process until a terminal node is reached. This procedure will select, with equal probability, any node corresponding to state-actions which have not been observed to lead to a penalty. Once a terminal node is reached, randomly and uniformly generate an action within the boundaries defined by that node.

Figure 4.4 gives an example of a penalty tree just after being initalized. For this example, assume the MDP has two state variables, x_1 and x_2, and one action variable, x_3. Discretize the state-action space by splitting each state variable once and the action variable twice. The number

in each node gives the value of the *penalty* variable. As no transitions have been observed yet, $penalty = 0$ for all nodes.

Now suppose the first transition resulting in a penalty has been observed. The state-action pair for this observation corresponds to the third node of the last level, and the system transitioned into a penalty. Figure 4.5 shows the tree at this point.

As more transitions are observed, eventually enough penalties will be observed to recognize that for some states, all actions lead to penalties. Figure 4.6 shows how the penalty tree might look when this happens for the first time. The node at level 3 colored red is now an artificial penalty, and any subsequent transitions into a state corresponding to that node will restart the exploration trajectory.

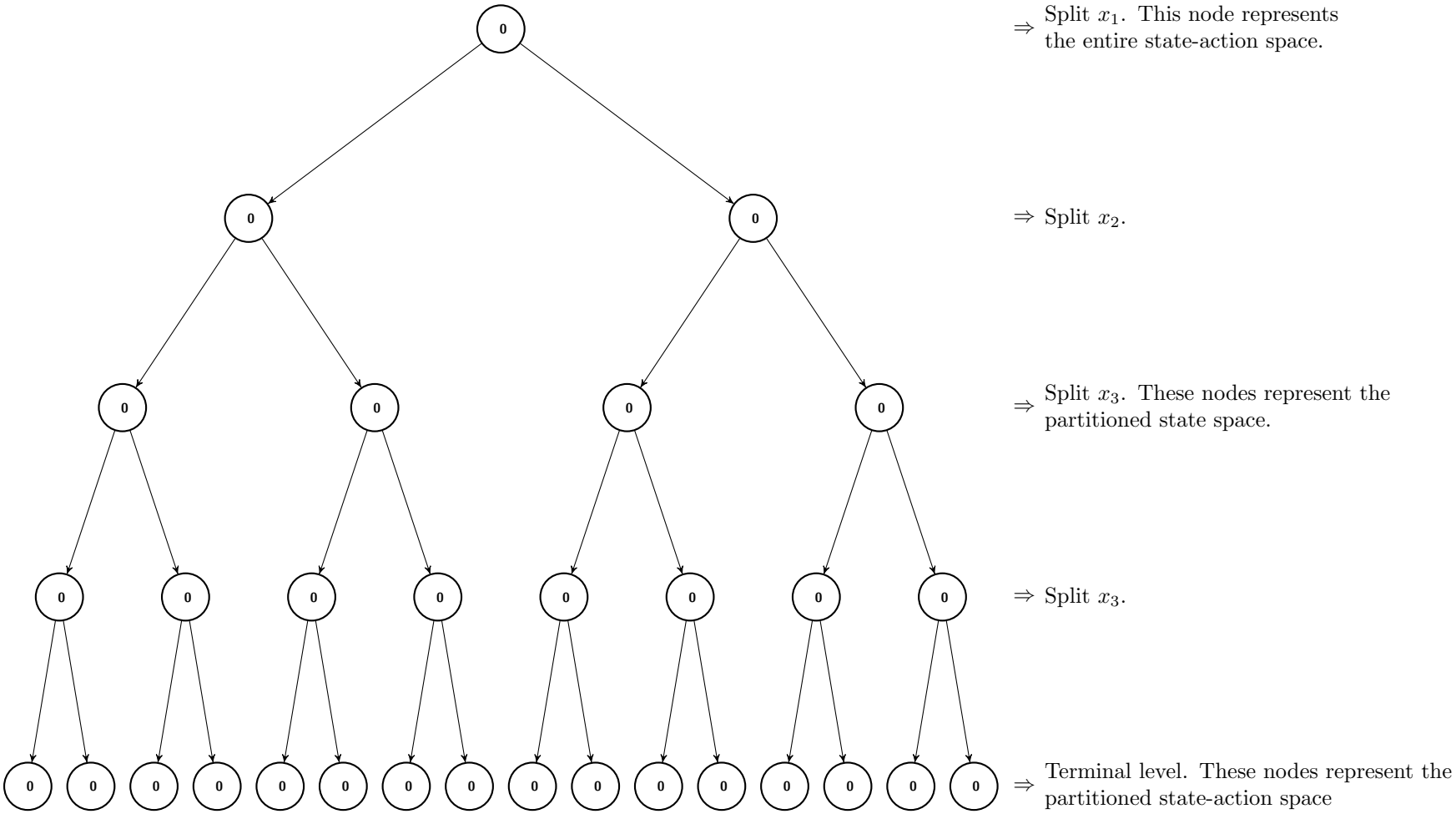

Figure 4.4: A penalty tree after being initialized. No penalties have been observed, so $penalty = 0$ for all nodes.

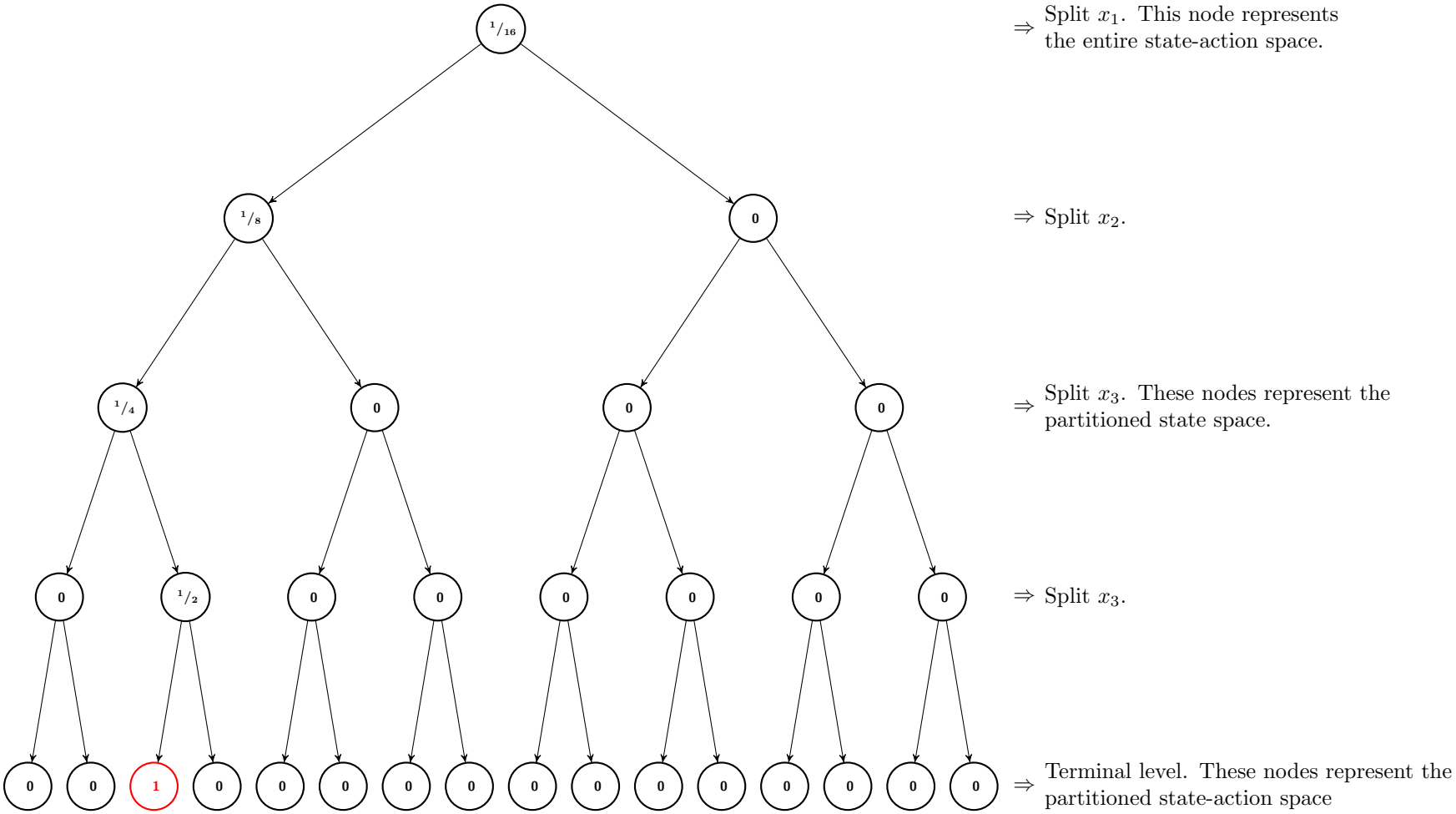

Figure 4.5: The penalty tree after seeing its first penalty. Random action selection now excludes the red node.

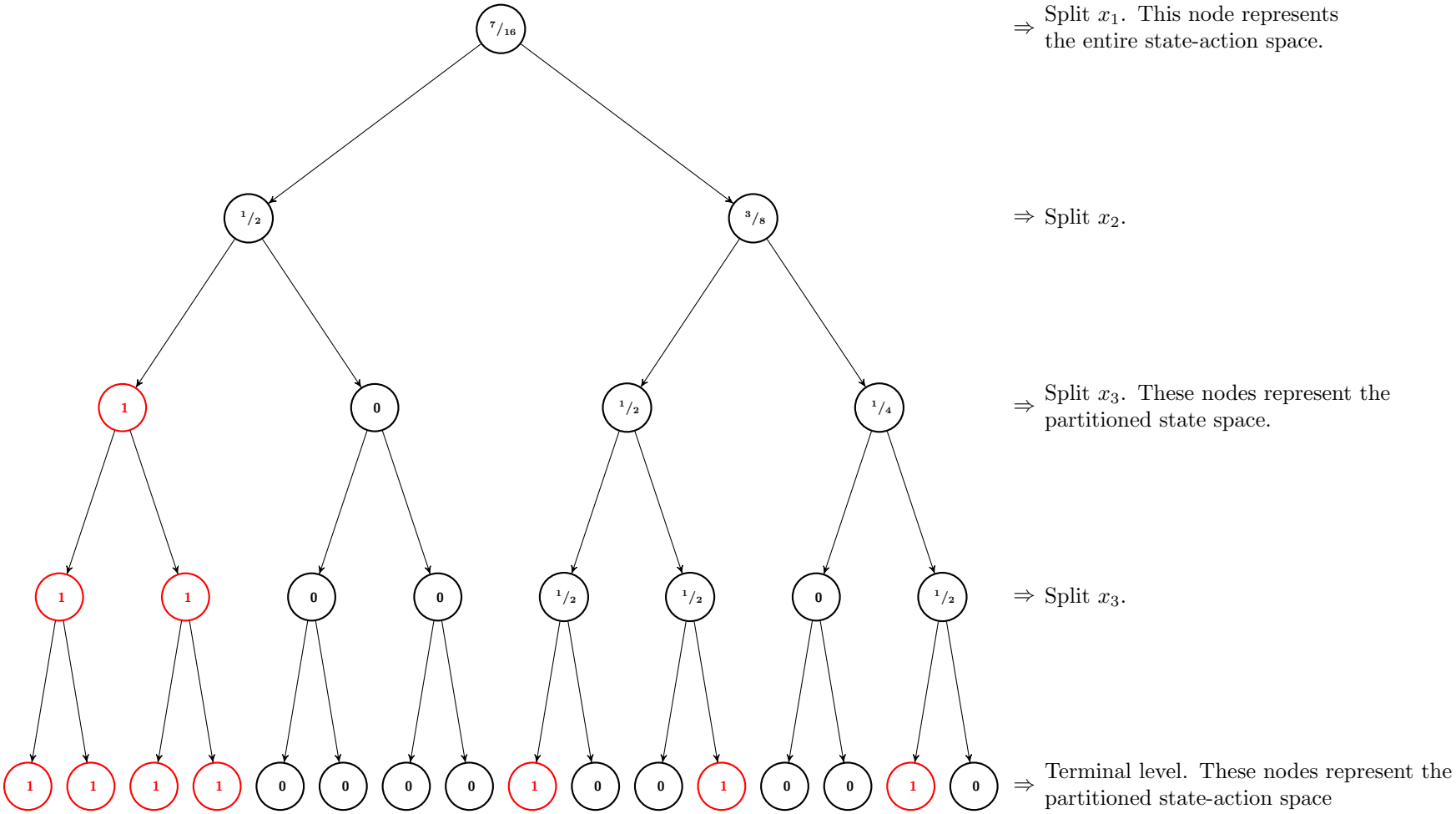

Figure 4.6: The penalty tree after observing enough penalties to classify its first artificial penalty.

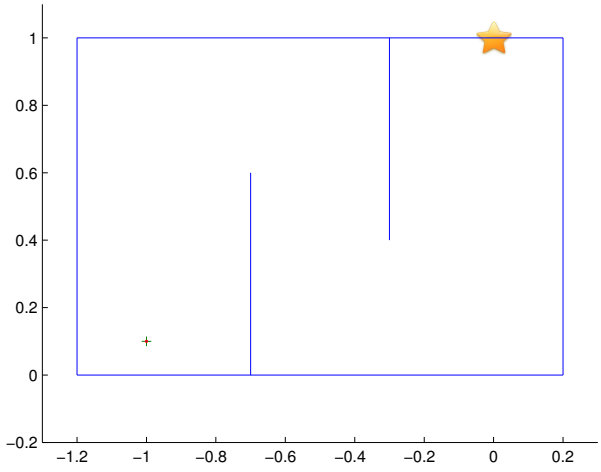

Figure 4.7: A diagram of the Zero-Gravity Rocket problem. The star indicates the goal and the cross in the lower left indicates the initial position.

4.3 The Zero-Gravity Rocket Problem

The Mountain Car and Inverted Pendulum problems are 3-dimensional, which is small enough to explore adequately with totally random action selection. Now consider a more difficult problem. A rocket loaded with explosives is in a spaceport and must find a docking area. The z-component of the rocket's initial position matches that of the dock, so the controller only need worry about maneuvering through 2-dimensional space. However, there are barriers between the initial position and the dock. Because of the explosive payload, any contact with the barriers will destroy the rocket and the trial restarts. The agent controls the rocket via thrusters on the front, back, and both sides of the rocket.

The state space is defined by four variables. x_p and y_p define the horizontal and vertical positions, and x_v and y_v define the horizontal and vertical velocities. The action space is defined by two variables. x_a and y_a are the horizontal and vertical accelerations. All together, this is a 6-dimensional problem with a very small probability of a randomly generated sequence of actions reaching the goal state. Figure 4.7 shows the boundaries and barriers as well as the initial position and goal location.

We will show that using APL for exploration is superior to random action selection both in terms of goal visits per trajectory and goal visits per time. Let us consider goal visits per

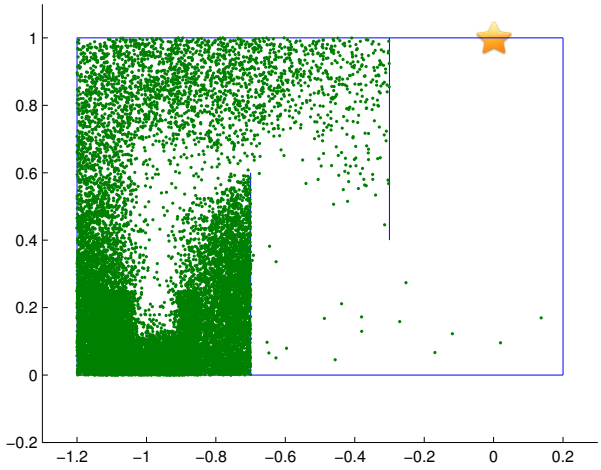

Figure 4.8: The end states for 25,000 trajectories generated by random action selection.

trajectory first. Figure 4.8 shows the last position (before transitioning into a penalty state) for 25,000 trajectories generated with random action selection. Notice that most of the trajectories end very early and none are close to reaching the goal.

Figure 4.9 shows the last positions for 25,000 trajectories generated using APL. Here, we expect the probability of getting close to the goal to increase over time. To see if this is true, the 25,000 trajectories are broken into 5 groups of 5,000 and plotted separately. The first plot shows the first 5,000 trajectories, the second plot shows the next set of 5,000 trajectories, and so on. It is clear that the distribution of positions is changing. The area with the most dense concentration of positions is under the initial state in the first group, moves up with the second, third, and fourth groups, and in the last group can be seen moving right towards the second barrier. Also, if one examines the position of the best trajectories (those closest to the goal), there is a slow but steady increase in the density of positions directly underneath the goal, and several individual trajectories which are quite close.

It would be informative to have a graphic illustrating how many state-actions have been marked as leading to penalties. Because the state space consists not only of positions but also velocities, the state space itself is difficult to illustrate. A work-around can be achieved by making a figure illustrating a subset of the state space. Begin by making a grid by discretizing the horizontal and vertical position variables. For each rectangle in the grid, search for all state-action terminal

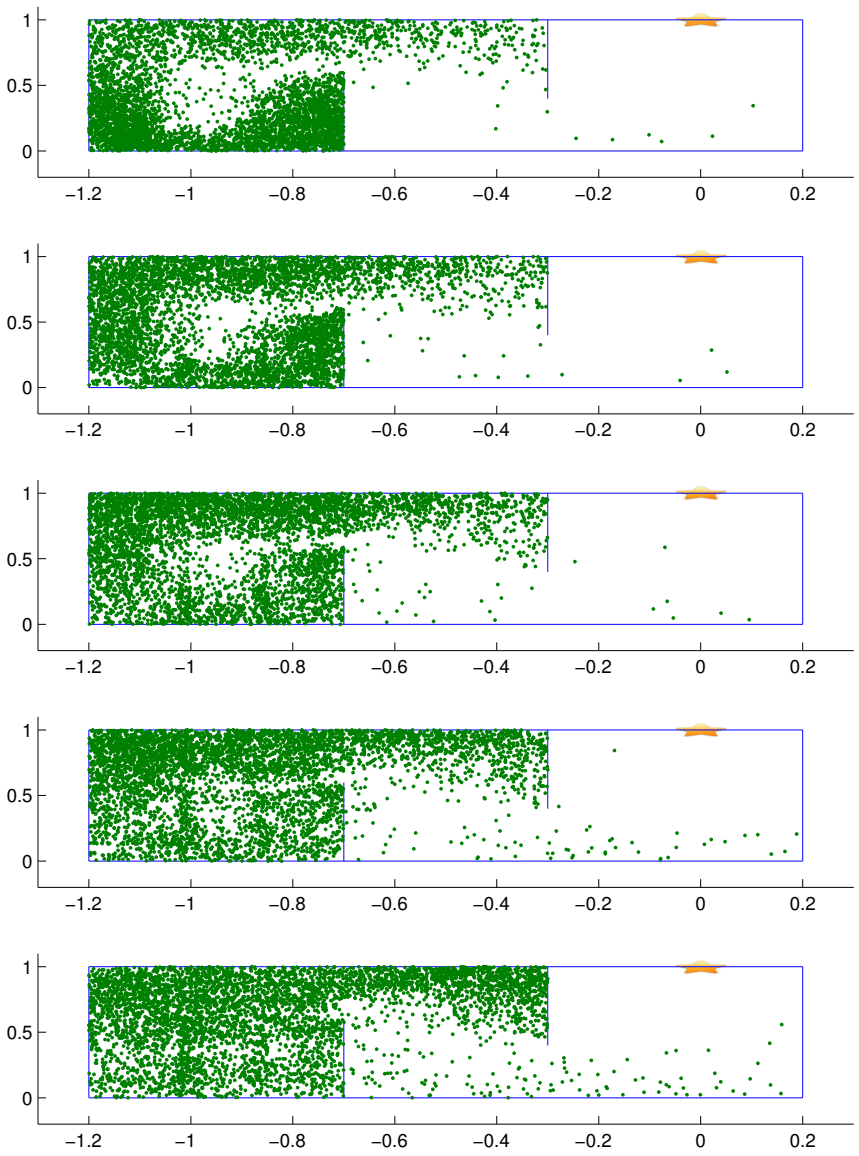

Figure 4.9: The end states for 25,000 trajectories when using APL. To illustrate the improving quality of trajectories, the trajectories are split into 5 groups of 5,000.

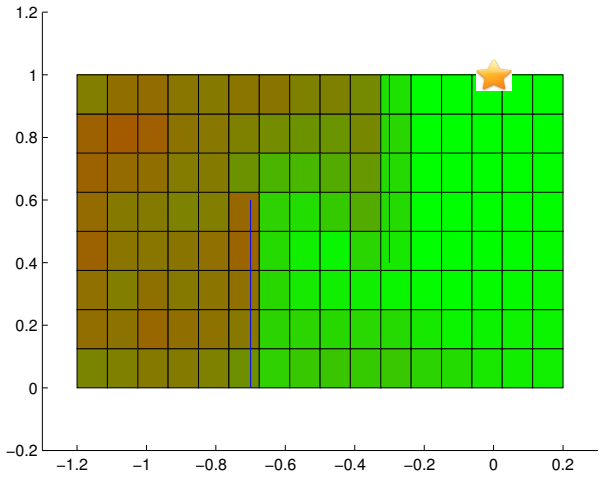

Figure 4.10: The color of each rectangle represents the proportion of corresponding state-actions that have been observed to lead to penalties. 25,000 transitions have been observed.

nodes corresponding to that rectangle. Calculate the proportion of those state-actions which have been marked as penalties, and project that proportion onto a color-scale with green meaning no penalties and red meaning all state-actions lead to penalties. Figure 4.10 gives such an illustration after 25,000 trajectories.

The preceding figures should make it clear that APL yields better average exploration given the same number of transitions. However, in application it is not ultimately the number of transitions that is of concern; it is the time needed to generate and process those transitions. One must consider the overhead cost of building and maintaining the penalty tree structure as well. To quantify the overhead, random exploration and APL exploration each ran for 500 seconds, and the time spent on various tasks was recorded. The results are in Table 4.1.

Task	**Random**	**APL**
Building Tree	-	.0824
Action Selection	113.845	226.2057
Calculating Transition	228.0396	59.7317
Checking for Penalty	-	152.1208
Updating Tree	-	16.4248
Other tasks	158.1154	45.4346
Transitions	12,935,738	2,899,852
Goals	6	237

Table 4.1: An analysis of time spent over 500 seconds for each exploration method.

There are a few interesting observations to be made. For one, the overhead of APL (consisting of the increased time in action selection, checking for penalties, and updating the penalty tree) is significant. Random exploration completes nearly 13 million transitions, compared with about 3 million for APL exploration. However, even though random exploration generates transitions roughly 4.5 times faster, it only finds 6 rewards compared to 237 for APL. Thus APL provides a double advantage over random exploration: APL takes less time to generate data with an adequate number of goal visits, and since it generates fewer transitions, the value-function learning phase of the algorithm will also take less time.

Another observation is how the overhead costs of APL break down. Even though the penalty tree structure is quite large at $2^{20} - 1 = 1048575$ nodes (compare to 12547 nodes for the regression tree that solved the Inverted Pendulum problem), it is constructed in less than a tenth of a second. This rapid construction is possible because of the highly structured layout of a penalty tree. A penalty tree is a fully grown binary tree, so by assigning a number to each node, every initial property of a node can be extracted from a binary representation of its assigned number. This allows for vectorized tree construction as opposed to constructing in a slower for-loop as is necessary for regression trees.

The entry labeled "Other tasks" consists nearly entirely of time-keeping. One may notice that time-keeping represents a much higher percentage of the total time for random exploration. This can be explained by noting that the time spent time-keeping is proportional to the number of transitions completed. Random exploration completes many more transitions and uses a small amount of computational time to record the time needed for that transition. In fact, because the Zero-Gravity Rocket has such simple mechanics, the time spent simulating a transition is only slightly larger than the time to measure and record the simulation time. For APL exploration, each transition also requires more complex action selection, checking for artificial penalties, and updating the tree if a penalty is found. This total time is significant compared to the time used to record how long it takes these tasks to complete, thus time-keeping is proportionally smaller for APL.

This leads to another interesting observation: because the problem considered has relatively simple mechanics (consider that the Mountain Car problem required the derivative of the hill in order to calculate gravitational force perpendicular to the hill, and the Inverted Pendulum has many trigonometric terms), the APL overhead is more significant for this problem than for many, perhaps most, others. Thus when testing whether APL is an improvement over random exploration, APL

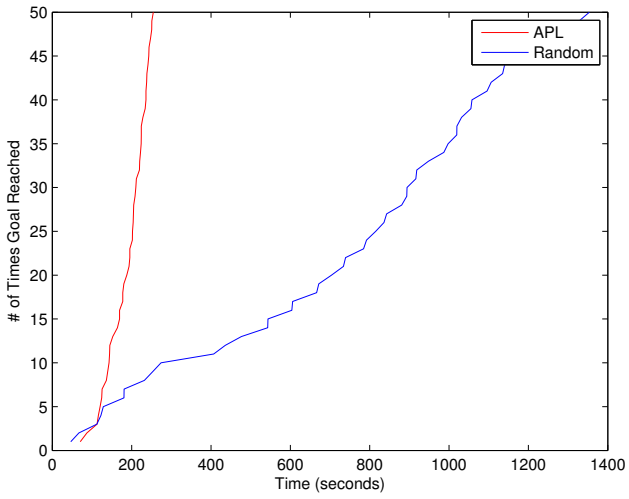

Figure 4.11: A plot showing how long it takes to reach the goal a given number of times. The blue line represents random action selection. The red line represents APL exploration.

is at a disadvantage. To quantify, for the Zero Gravity Rocket problem, APL processes transitions about a fifth as fast as random exploration. A problem with more complicated dynamics, which would take longer to simulate transitions, could see APL processing transitions only half as fast as random exploration. The recognition that APL is being compared in a disadvantageous scenario makes its superior performance even more impressive.

It is clear from Table 4.1 that APL finds the goal more often than random exploration, but the times at which the goal is found is not clear. Figure 4.11 plots the times at which the goal is visited until 50 goal visits have been seen. The blue line represents random exploration, and is roughly linear as one would expect. The red line represents APL, and is increasing at a super-linear rate.

One could continue the analysis of random versus APL exploration further by continuing into the value-function learning phase and measuring the time needed to arrive at a near-optimal solution, but this is unnecessary. The value-function learning phase will proceed similarly whether the data was generated from random action selection or APL. The only differences will be APL explores the state-action space more uniformly, visits the goal state more often, and indirectly speeds up value-function learning by passing less transition data to be processed.

Chapter 5

Conclusion

This dissertation has presented research into the theory and application of RL algorithms when states and actions are continuous. Chapter 2, the principal result, presented an algorithm based on kernel regression and proved convergence to an optimal policy. This result should be understood to be a chiefly theoretical contribution, and not intended to compete with the state-of-the-art.

Chapters 3 and 4 are less theoretical and more practical. Chapter 3 introduced regression trees and detailed modifications for use in continuous action RL. Experimental results were provided to show evidence of superior computational performance over the algorithm of Chapter 2. Chapter 4 discussed difficulties with random exploration and introduced penalty trees, data structures for learning how to avoid penalties before the value-function learning phase begins.

5.1 Future Work

Concerning the kernel based algorithm of Chapter 2, this research can be regarded as complete. The author's desire was to create the the first online Q-learning variant for continuous actions proven to converge to an optimal policy, and this goal is complete. One possible extension concerns the experimental section. Selecting optimal actions is possible due to special properties of the kernel used and the single action variable. Extending this strategy to arbitrary kernels or multiple action variables is a possible direction, but given the far superior performance of non-kernel based methods, this has limited value.

In the experiments of Chapter 3, unexpected interactions between components of the re-

gression trees were observed. A better understanding of these interactions and how to choose the best tree variant for a specific problem could be worthwhile investigations. Also, the variants using linear regression for predicting state-action values performs less well than one might expect. One could test whether custom-coded functions (rather than using MATLAB's built-in functions) could improve performance enough to make those variants worth using.

Chapter 4 saw a direct comparison of totally random exploration and APL. While totally random exploration is the only option for pure batch algorithms, it is possible to construct algorithms that are hybrids of pure batch and pure sequential. Such an algorithm will produce a first (relatively small) batch by totally random exploration, pass this small sample to the value-function phase, then use the learned value-function to guide exploration and produce another sample. This repeats until the sample size is large enough or the learned policy solves the problem satisfactorily. This hybrid approach is known as *growing batch*. Showing that APL is superior to growing batch as well as pure batch is an obvious next step.

Encountering a penalty is not the only undesirable behavior APL could learn to avoid. For example, in a deterministic goal-driven problem like Mountain Car or Zero-Gravity Rocket, an action which does not result in a change of state will clearly not bring the agent closer to the goal. APL could be modified relatively easily to check for stagnation, and to mark state-actions which do not lead to a new state as a penalty. Also, for both problems mentioned, visiting the same state twice in a single trajectory indicates a loop. If the algorithm could recognize and mark as a penalty the state-action which resulted in the loop this would further improve exploration.

More generally, the success of APL demonstrates the success of a limited form of learning prior to engaging in full-fledged value-function learning. APL is focused entirely on learning the penalty structure, and thus is well-suited to problems like the Inverted Pendulum and Zero-Gravity Rocket problems. For problems where penalties play a less significant role, like the Mountain Car, the benefit of APL will be lessened. Are there other ways to structure limited learning that are not based on penalties?

As mentioned in Section 1.3, if one is willing to break with the custom of generating transitions along continuous trajectories, quasi-random sequences can be used to generate state-actions uniformly with low discrepancy. Quasi-random sequences could still be used to drive exploration restricted to continuous trajectories. Consider an MDP with finite states and continuous actions. By assigning an action-valued quasi-random sequence to each state and choosing actions as a function

of the sequence and the number of visits to the state, more uniform exploration would be achieved. Now consider continuous states and continuous actions. Is it possible to construct a function which can assign actions to states using the number of visits to states as input, with the actions for each state forming a low-discrepancy sequence? Such a sequence could guarantee visiting a goal within a finite amount of time, and intuition says the time would be shorter than the expected time until finding the goal with totally random exploration. Finding such a function and verifying intuition is the most likely next project.

Finally, with regards to future research, it would be nice to move in the direction of more traditional statistics. RL is a fun field to work in and has opportunities for some interesting regression applications, but a significant portion lies in the realm of computer science. Chapters 3 and 4 in particlar, though data driven, begin to stray to the boundaries between mathematics and computer science.

Appendix A Quasi-random Sequences

Section 1.3 suggests exploring by choosing states and actions from a quasi-random sequence in order to ensure uniform exploration, and Section 3.5 describes an experiment with quasi-randomly generated transition data. This appendix will explain what is meant by a quasi-random sequence and how to generate one. The definitions, theorems, and exposition is based on the text by Kuipers and Niederreiter [16].

Let $(x_n), n = 1, 2, \cdots$ be a sequence of real numbers taking values in the unit interval $I = [0, 1]$. For a positive integer N and a subset E of I, let the counting function $A_N(E)$ be defined as the number of terms $x_n, 1 \leq n \leq N$, for which $x_n \in E$.

Definition The sequence (x_n) is *uniformly distributed* if for every pair a, b of real numbers with $0 \leq a < b \leq 1$,

$$\lim_{N \to \infty} \frac{A_N([a, b))}{N} = b - a. \tag{1}$$

The obvious example of a uniformly distributed sequence is a sequence of uniform random variables, whether truly random or generated pseudo-randomly by a computer. Such a sequence will have areas of high and low density. Consider subintervals of identical width $[\frac{i}{M}, \frac{i+1}{M})$ for $i = 0, 1, \cdots, M-1$ for large M. It may take an extremely long time before each subinterval is visited, whereas other subintervals will have many visits.

For many situations, it would be advantageous to have a sequence for which Equation (1) converges quickly. This motivates the next definition.

Definition Given a sequence (x_n) and finite N, the *discrepancy* of a sequence up to index N, denoted D_N, is defined by

$$D_N = \sup_{0 \leq a \leq b \leq 1} \left| \frac{A_N([a, b))}{N} - (b - a) \right|. \tag{2}$$

It can be proved (see Theorem 1.1 of Chapter 2 in the above referenced text) that a sequence is uniformly distributed if and only if $\lim_{N \to \infty} D_N = 0$. The discrepancy gives information about how quickly a sequence achieves uniform density. A sequence is called *quasi-random* if it is uniformly distributed and has low discrepancy.

A.1 Van der Corput Sequences

The Van der Corput sequence given a prime p is a quasi-random sequence which is easy to generate, has low discrepancy, and is easy to generalize to multiple dimensions. It is constructed by converting the natural numbers into base p, reflecting the digits across the decimal, and converting back into base 10. Table A.1 shows the construction of the first 7 entries of the Van der Corput sequence for $p = 2$.

Natural number	**Base p representation**	**Reflected**	**Sequence entry**
1	1	.1	.5
2	10	.01	.25
3	11	.11	.75
4	100	.001	.125
5	101	.101	.675
6	110	.011	.375
7	111	.111	.875

Table A.1: The first 7 entries of the Van der Corput sequence for $p = 2$.

The discrepancy of the Van der Corput sequence for $p = 2$ satisfies

$$D_N \leq \frac{\log(N)}{N \log 2}$$

which is the lowest rate for all known sequences.

A.2 Halton Sequences

A *Halton sequence* is a multi-dimensional quasi-random sequence. Each component of an entry in a Halton sequence comes from a Van der Corput sequence of a different prime. For an example of dimension 2, let (x_n) and (y_n), $n = 1, 2, \cdots$ be Van der Corput sequences constructed with primes p_1 and p_2 respectively, $p_1 \neq p_2$. Construct a Halton sequence by setting entry n equal to (x_n, y_n).

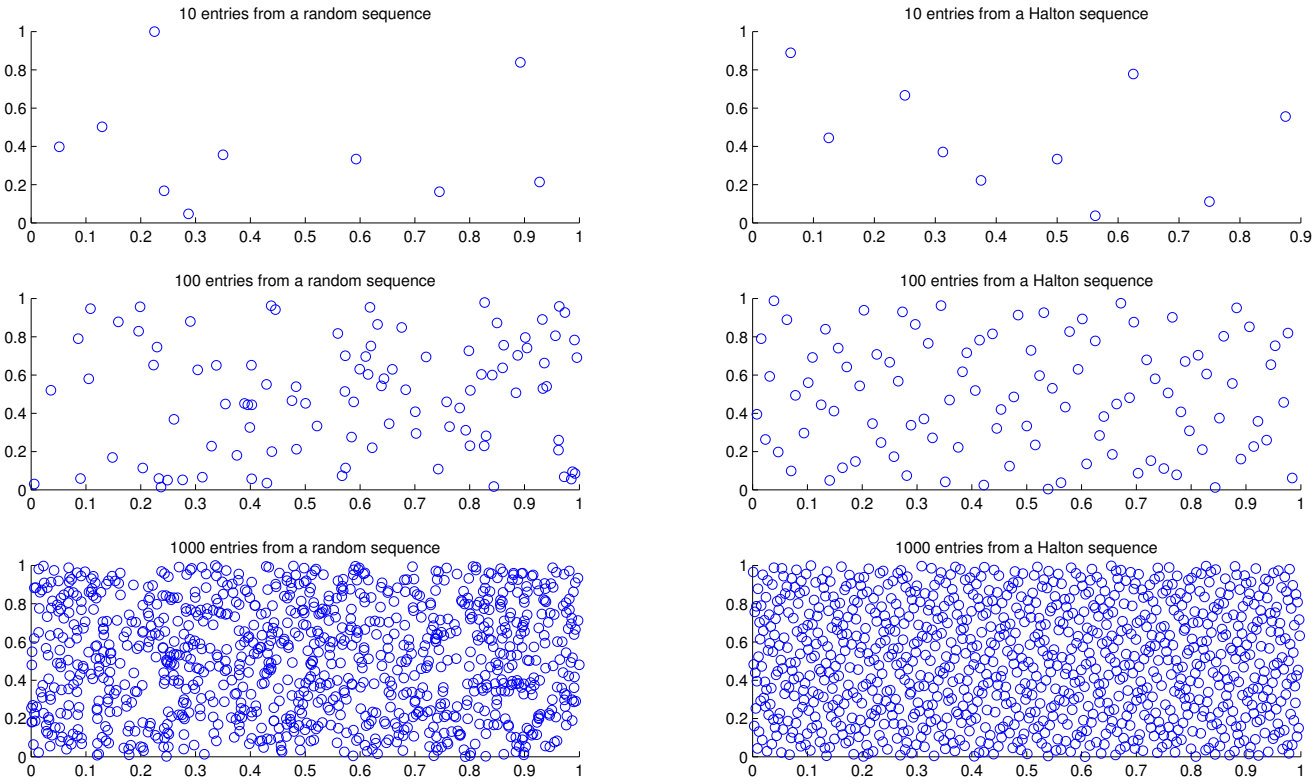

Figure A.1: Comparisons of random and quasi-random two-dimensional sequences.

A.3 Code

```
function sequence = vandercorput(p,n)
% SUMMARY: Generates the first n entries of the Van der Corput
%   sequence given prime p.
% INPUT: p is a prime number.
%        n is the number of entries to generate
% OUTPUT: sequence is the generated quasi-random sequence


sequence = zeros(1,n);


parfor k=1:n
```

```
    result = 0;
    f=1/p;
    i=k;
    while i>0
        result = result + f*mod(i,p);
        i = floor(i/p);
        f = f/p;
    end
    sequence(k) = result;
end


function sequence = three_dim_halton(p_1, p_2, p_3, n)
% SUMMARY: Generates a three dimensional Halton sequence
%   using primes p_1, p_2, p_3.
% INPUT: p_1, p_2, and p_3 are the primes to generate each
%   Van der Corput sequence.
%        n is length of sequence to be generated
% OUTPUT: sequence is the multidimensional quasi-random
%    sequence

sequence = cell(1,n);
sequence_1 = vandercorput(p_1, n);
sequence_2 = vandercorput(p_2, n);
sequence_3 = vandercorput(p_3, n);

parfor k = 1:n
    sequence{k} = [sequence_1(k), sequence_2(k), sequence_3(k)];
end
```

Bibliography

[1] J. S. Albus. A new approach to manipulator control: the cerebellar model articulation controller (CMAC). *Journal of Dynamic Systems, Measurement, and Control*, 97:220–227, 1975.

[2] Leemon C. Baird and A. Harry Klopf. Reinforcement learning with high-dimensional, continuous actions. Technical Report WL–TR-93-1147, Wright-Patterson Air Force Base Ohio: Wright Laboratory, 1993.

[3] Dimitri P. Bertsekas. A counterexample to temporal differences learning. *Neural Computation*, 7(2):270–279, March 1995.

[4] Patrick Billingsley. *Convergence of probability measures*. Wiley Series in Probability and Statistics: Probability and Statistics. John Wiley & Sons Inc., New York, second edition, 1999. A Wiley-Interscience Publication.

[5] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth International Group, Belmont, CA, 1984.

[6] John M. Chambers. Regression updating. *Journal of the American Statistical Association*, 66(336):pp. 744–748, 1971.

[7] L. Devroye and L. Györfi. *Nonparametric density estimation: the L1 view*. Wiley series in probability and mathematical statistics. Wiley, 1985.

[8] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6:503–556, 2005.

[9] Michael Fairbank and Eduardo Alonso. The divergence of reinforcement learning algorithms with value-iteration and function approximation. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, 2012.

[10] Chris Gaskett, David Wettergreen, and Alexander Zelinsky. Q-learning in continuous state and action spaces. In *Australian Joint Conference on Artificial Intelligence*, pages 417–428. Springer-Verlag, 1999.

[11] Geoffrey J. Gordon. Chattering in SARSA(lambda) - a CMU learning lab internal report. Technical report, Carnegie Mellon University, 1996.

[12] Bruce E. Hansen. Uniform convergence rates for kernel estimation with dependent data. *Econometric Theory*, 2008.

[13] W. Härdle. *Nonparametric and Semiparametric Models*. Springer Series in Statistics. Springer Berlin Heidelberg, 2004.

[14] Tommi Jaakkola, Michael I. Jordan, and Satinder P. Singh. Convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201, 1994.

[15] J. Jacod and P.E. Protter. *Probability Essentials*. Universitext (1979). Springer, 2003.

[16] Lauwerens Kuipers and Harald Niederreiter. *Uniform Distribution of Sequences*. Wiley, 1974.

[17] Alessandro Lazaric, Marcello Restelli, and Andrea Bonarini. Reinforcement learning in continuous action spaces through sequential Monte Carlo methods. In *Adv. Neural Information Proc. Systems*, 2007.

[18] J. R. Millán, D. Posenato, and E. Dedieu. Continuous-action Q-learning. *Machine Learning*, 2002.

[19] Andrew Moore. *Efficient Memory-based Learning for Robot Control*. PhD thesis, Robotics Institute, Carnegie Mellon University, March 1991.

[20] E. Nadaraya. On estimating regression. *Theory of Probability and its Applications*, 9:141–142, 1964.

[21] Dirk Ormoneit and Saunak Sen. Kernel-based reinforcement learning. *Machine Learning*, 1999.

[22] S.M. Ross. *Applied Probability Models with Optimization Applications*. Dover Books on Mathematics. Dover Publications, 1992.

[23] G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical report, Cambridge University Engineering Department, 1994.

[24] Juan Carlos Santamaría, Richard S. Sutton, and Ashwin Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavor*, 1996.

[25] B. W. Silverman. *Density estimation for statistics and data analysis*. Chapman and Hall, London, 1986.

[26] J.S. Simonoff. *Smoothing methods in statistics*. Springer, New York, 1996.

[27] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[28] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pages 1038–1044. MIT Press, 1996.

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

[30] Csaba Szepesvári and William D. Smart. Interpolation-based Q-learning. In *Proceedings of the International Conference on Machine Learning*, pages 791–798. ACM Press, 2004.

[31] Gavin Taylor and Ronald Parr. Kernelized Value Function Approximation for Reinforcement Learning. In *Proceedings of the 26th International Conference on Machine Learning*, pages 1017–1024, 2009.

[32] Stephan H.G. ten Hagen. *Continuous State Space Q-Learning for Control of Nonlinear Systems*. PhD thesis, University of Amsterdam, 2001.

[33] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Commun. ACM*, 38(3):58–68, 1995.

[34] John N. Tsitsiklis. Asynchronous stochastic approximation and Q-learning. In *Machine Learning*, pages 185–202, 1994.

[35] John N. Tsitsiklis and Benjamin Van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22(1–3):59–94, March 1996.

[36] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, 1989.

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

[38] Geoffrey S. Watson. Smooth regression analysis. *Sankhya: The Indian Journal of Statistics*, 26:359–372, 1964.

[39] Xin Xu, Dewen Hu, and Xicheng Lu. Kernel-based least squares policy iteration for reinforcement learning. *IEEE Transactions on Neural Networks*, 18(4):973–992, 2007.