Reinforcement Learning Through Gradient Descent

Leemon C. Baird III

May 14, 1999
CMU-CS-99-132

School of Computer Science
Carnegie Mellon Universi9ty
Pittsburgh, PA 15213

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis committee:
Andrew Moore (chair)
Tom Mitchell
Scott Fahlman
Leslie Kaelbling, Brown University

This research was supported in part by the U.S. Air Force, including the Department of Computer Science, U.S. Air Force Academy, and Task 2312R102 of the Life and Environmental Sciences Directorate of the United States Office of Scientific Research. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Air Force Academy, U.S. Air Force, or the U.S. government

Keywords: Reinforcement learning, machine learning, gradient descent, convergence, backpropagation, backprop, function approximators, neural networks, Q-learning, TD(lambda), temporal difference learning, value function approximation, evaluation functions, dynamic programming, advantage learning, residual algorithms, VAPS

Abstract

Reinforcement learning is often done using parameterized function approximators to store value functions. Algorithms are typically developed for lookup tables, and then applied to function approximators by using backpropagation. This can lead to algorithms diverging on very small, simple MDPs and Markov chains, even with linear function approximators and epoch-wise training. These algorithms are also very difficult to analyze, and difficult to combine with other algorithms.

A series of new families of algorithms are derived based on stochastic gradient descent. Since they are derived from first principles with function approximators in mind, they have guaranteed convergence to local minima, even on general nonlinear function approximators. For both *residual* algorithms and *VAPS* algorithms, it is possible to take any of the standard algorithms in the field, such as Q-learning or SARSA or value iteration, and rederive a new form of it with provable convergence.

In addition to better convergence properties, it is shown how gradient descent allows an inelegant, inconvenient algorithm like Advantage updating to be converted into a much simpler and more easily analyzed algorithm like Advantage learning. In this case that is very useful, since Advantages can be learned thousands of times faster than Q values for continuous-time problems. In this case, there are significant practical benefits of using gradient-descent-based techniques.

In addition to improving both the theory and practice of existing types of algorithms, the gradient-descent approach makes it possible to create entirely new classes of reinforcement-learning algorithms. VAPS algorithms can be derived that ignore values altogether, and simply learn good policies directly. One hallmark of gradient descent is the ease with which different algorithms can be combined, and this is a prime example. A single VAPS algorithm can both learn to make its value function satisfy the Bellman equation, and also learn to improve the implied policy directly. Two entirely different approaches to reinforcement learning can be combined into a single algorithm, with a single function approximator with a single output.

Simulations are presented that show that for certain problems, there are significant advantages for Advantage learning over Q-learning, residual algorithms over direct, and combinations of values and policy search over either alone. It appears that gradient descent is a powerful unifying concept for the field of reinforcement learning, with substantial theoretical and practical value.

Acknowledgements

I thank Andrew Moore, my advisor, for great discussions, stimulating ideas, and a valued friendship. I thank Leslie Kaelbling, Tom Mitchell, and Scott Fahlman for agreeing to be on my committee, and for their insights and help. It was great being here with Kwun, Weng-Keen, Geoff, Scott, Remi, and Jeff. Thanks. I'd like to thank CMU for providing a fantastic environment for doing research. I've greatly enjoyed the last three years. A big thanks to my friends Mance, Scott, Brian, Bill, Don and Cheryl, and especially for the support and help from my family, from my church, and from the Lord.

This research was supported in part by the U.S. Air Force, including the Department of Computer Science, U.S. Air Force Academy, and Task 2312R102 of the Life and Environmental Sciences Directorate of the United States Office of Scientific Research. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Air Force Academy, U.S. Air Force, or the U.S. government

Contents

Figures

Tables

1 Introduction

Reinforcement learning is a field that can address a wide range of important problems.

Optimal control, schedule optimization, zero-sum two-player games, and language learning are all problems that can be addressed using reinforcement-learning algorithms.

There are still a number of very basic open questions in reinforcement learning, however. How can we use function approximators and still guarantee convergence? How can we guarantee convergence for these algorithms when there is hidden state, or when exploration changes during learning? How can we make algorithms like Q-learning work when time is continuous or the time steps are small? Are value functions good, or should we just directly search in policy space?

These are important questions that span the field. They deal with everything from low-level details like finding maxima, to high-level concepts like whether we should be even using dynamic programming at all. This thesis will suggest a unified approach to all of these problems: gradient descent. It will be shown that using gradient descent, many of the algorithms that have grown piecemeal over the last few years can be modified to have a simple theoretical basis, and solve many of the above problems in the process. These properties will be shown analytically, and also demonstrated empirically on a variety of simple problems.

Chapter 2 introduces reinforcement learning, Markov Decision Processes, and dynamic programming. Those familiar with reinforcement learning may want to skip that chapter. The later chapters briefly define some of the terms again, to aid in selective reading.

Chapter 3 reviews the relevant known results for incremental and stochastic gradient descent, and describes how these theorems can be made to apply to the algorithms proposed in this thesis. That chapter is of theoretical interest, but is not needed to understand the algorithms proposed. The proposed algorithms are said to converge "in the same sense that backpropagation converges", and that chapter explains what this means, and how it can be proved. It also explains why two independent samples are necessary for convergence to a local optimum, but not for convergence in general.

Chapters 4, 5, and 6 present the three main algorithms: Residual algorithms, Advantage learning, and VAPS. These chapters are designed so they can be read independently if there is one algorithm of particular interest. Chapters 5 and 6 both use the ideas from chapter 4, and all three are based on the theory presented in chapter 3, and use the standard terminology defined in chapter 2.

Chapter 4 describes residual algorithms. This is an approach to creating pure gradient-descent algorithms (called *residual gradient* algorithms), and then extending them to a larger set of algorithms that converge faster in practice (called *residual* algorithms). Chapters 5 and 6 both describe residual algorithms, as proposed in chapter 4.

Chapter 5 describes *Advantage learning*, which allows reinforcement learning with function approximation to work for problems in continuous time or with very small time steps. For MDPs with continuous time (or small time steps) where Q functions are preferable to value functions, this algorithm can be of great practical use. It is also a residual algorithm as defined in chapter 4, so it has those convergence properties as well.

Chapter 6 describes *VAPS*, which allows the exploration policy to change during learning, while still giving guaranteed convergence. In addition, it allows pure search in policy space, learning policies directly without any kind of value function, and even allows the two approaches to be combined. VAPS is a generalization of residual algorithms, as described in chapter 4, and achieves the good theoretical convergence properties described in chapter 3. The VAPS form of several different algorithms is given, including the Advantage learning algorithm from chapter 5. Chapter 6 therefore ties together all the major themes of this thesis. If there is only time to read one chapter, this might be the best one to read.

Chapter 7 is a brief summary and conclusion.

2 Background

This chapter gives an overview of reinforcement learning, Markov Decision Processes and dynamic programming. It defines the standard terminology of the field, and the notation to be used throughout this thesis.

2.1 RL Basics

Reinforcement learning is the problem of learning to make decisions that maximize rewards or minimize costs over a period of time. The environment gives an overall, scalar reinforcement signal, but doesn't tell the learning system what the correct decisions would have been. The learning system therefore has much less information than in *supervised learning*, where the environment asks questions, and then tells the learning system what the right answers to those questions would have been. Reinforcement learning does use more information than *unsupervised learning*, where the learning system is simply given inputs and is expected to find interesting patterns in the inputs with no other training signal. In many ways, reinforcement learning is the most difficult problem of the three, because it must learn by trial and error from a reinforcement signal that is not as informative as might be desired.

This training signal typically gives *delayed reward*: a bad decision may not be punished until much later, after many other decisions have been made. Similarly, a good decision may not yield a reward until much later. Delayed reward makes learning much more difficult.

The next three sections define the three types of reinforcement learning problems (Markov chains, MDPs and POMDPs), and the two approaches to solving them (pure policy search, and dynamic programming).

2.1.1 Markov Chains

A *Markov chain* is a set of states $\mathbf{X}$, a starting state $x_0 \in \mathbf{X}$, a function giving transition probabilities, $P(x_t,x_{t+1})$, and a reinforcement function $R(x_t,x_{t+1})$. The state of the system starts in x_0. Time is discrete, and if the system is in state x_t at time t, then at time t+1, with probability $P(x_1,x_2)$, it will be in state x_{t+1}, and will receive reinforcement $R(x_t,x_{t+1})$. There are no decisions to make in a Markov chain, so the learning system typically tries to predict future reinforcements. The *value* of a state is defined to be the expected discounted sum of all future reinforcements:

$$V(x_t) = E\left[\sum_{i=t}^{\infty} \gamma^i R(x_i, x_{i+1})\right]$$

where $0 \leq \gamma \leq 1$ is a discount factor, and $E[]$ is the expected value over all possible trajectories. If a state transitions back to itself with probability 1, then the reinforcement is usually defined to be zero for that transition, and the state is called an *absorbing state*. If $\gamma=1$, then the problem is said to be *undiscounted*. If the reinforcements are bounded, and either $\gamma<1$ or all trajectories lead eventually to absorbing states with probability 1, then V is well defined.

Markov chains are rarely useful reinforcement learning problems by themselves, but are useful for solving more general problems. Here is one case, though, where the value of a state in a Markov chain has a useful meaning: suppose the time step in the chain represents one year, and the reinforcement represents the number of dollars that a share in a certain stock will pay each year in dividends. The chain always reaches an absorbing state with probability 1, representing the company going bankrupt. An investor has some money to invest for at least that long, and has the choice between either investing in that stock (and never selling it), or putting the money into a savings account with an interest rate of $((1/\gamma)-1)$, compounded annually. If the state right now is x_t, how much should the investor be willing to pay for one share of the stock? The answer is $V(x_t)$, as defined above.

This example illustrates what discounting does. If γ is close to one, then reinforcement in the distant future is almost as desirable as immediate reinforcement. If γ is close to zero, then only reinforcement in the near term matters much. So γ can be thought of as directly related to calculations for the present value of money in economics.

Another way to look at V is as a weighted sum of future reinforcements, where the first reinforcement has weight 1, the second has weight γ, the third has weight γ^2, and so on. How many terms does it take before half of the total weight has occurred? In other words, what is the "half-life" of this exponential weighting? The answer is $\text{Log}_\gamma 2$ steps. This is easy to remember to one significant figure for certain common values of γ. When $\gamma=0.9$, half of the reinforcement that matters happen in the first 7 time steps. When $\gamma=0.99$, the half-life is 70 time steps. For $\gamma=0.999$ it's 700, and for $\gamma=0.9999$ it's 7000. These rough numbers are useful to remember when picking a discount factor for a new reinforcement-learning problem.

2.1.2 MDPs

Markov chains are of limited interest because there are no decisions to make. Instead, most reinforcement learning deals with Markov Decision Processes (MDPs). An MDP is like a Markov chain, except that on every time step the learning system must look at the current state and choose an action from a set of legal actions. The transition probability and reinforcement received are then functions of both the state and the action. Given a discount factor γ, the problem is to learn a good *policy*, which is a function that picks an action in each state. When following a given policy, an MDP reduces to a Markov chain. The goal is to find the policy such that the resulting chain has as large a value in the start state as possible.

This is a very general problem. A control problem, such as flying a plane, can be viewed as an MDP, where the current position, attitude, and velocity make up the state, and the signals sent to the control surfaces constitute the action. Reinforcement might be a signal such as a 1 on every time step until the plane crashes, then a 0 thereafter. That is equivalent to telling the learning system to do whatever it takes to avoid crashes, but not giving it any clues as to what it did wrong when it does crash.

Optimization problems can also be thought of as MDPs. For example, to optimize a schedule that tells several shops what jobs to do and in what order, you can think of a completely-filled-out schedule as being a state. An action is then the act of making a change to the schedule. The reinforcement would be how good the schedule is, optimizing speed or cost or both.

2.1.3 POMDPs

In an MDP, the next state is always a stochastic function of the current state and action. Given the current state and action, the next state is independent of any previous states and actions. This is the *Markov property*, and systems without that property are called *Partially Observable Markov Decision Processes* (POMDPs). An example would be a the card game Blackjack, where the probability of the next card drawn from the deck being an ace is not just a function of the cards currently visible on the table. It is also a function of how many aces have already been drawn from the current deck. In other words, if the "state" is defined to be those cards that are currently visible, then the probability distribution of the next state is not just a function of the current state. It is also a function of previous states.

There is a simple method for transforming any POMDP into an MDP. Just redefine the "state" to be a list of all observations seen so far. In Blackjack, the current "state" would be a record of everything that has happened since the last time the deck was shuffled. With that definition of state, the probability distribution for the next card truly is a function only of the current state, and not of previous states. Unfortunately, this means that number of states will be vastly increased, and the dimensionality of the state space will change on each step, so this may make solving the problem difficult.

Another approach to converting a POMDP to an MDP is the *belief state* approach. This is applicable when the POMDP is a simple MDP with part of the state not visible. The agent maintains a probability distribution of what the non-observable part of the state is, and updates it according to Bayes rule. If you call this probability distribution itself the "state", then the POMDP is reduced to an MDP. This can be a much better approach than just recording a history of all observations, since the belief state is typically finite dimensional. In addition, this approach doesn't waste time remembering useless information. In the Blackjack example, a belief-state approach would simply remember which cards have been seen already, but would not record what order they were seen in, and would not record what actions were performed earlier.

Finally, there is an unfortunate term that has led to widespread confusion. A POMDP is often said to have *hidden state*, in contrast to an MDP, which does not. This reflects that most POMDPs can be thought of as MDPs where part of the state is not observable. However, that does *not* mean that any MDP with unobservable state will become a POMDP! For example, suppose a diver picks up a clam from the ocean floor. The diver does not know whether the clam contains a pearl. That is one aspect of the state of the universe that is not observable. It is also highly relevant to the diver's behavior: if there is no pearl, then it may not be worth the effort to open the clam. Since the state is hidden, and is important, does this become a POMDP? No. It is true that given the current observations, the diver cannot tell whether the pearl is there. However, remembering previous observations gives no more information than just the current observation. Therefore, it is not a POMDP. The system is still an MDP, despite the state that is hidden. The question to ask is always "would the agent be able to improve performance by remembering previous observations?". If the answer is yes, then it is a POMDP, otherwise it is an MDP.

2.1.4 Pure Policy Search

Given an MDP or POMDP, how can an agent find a good policy? The most straightforward approach is to make up a policy, evaluate it by following it for a while, then make changes to it. This pure policy search is the approach followed by genetic algorithms, backpropagation through time, and learning automata. This would be expected to work well for problems where local minima in policy space are rare. It would also be expected to work well when the number of policies is small compared to the size of the state space. For example, if there are two flight control programs that were written for the space shuttle, and there's only room for one, then there are really only two policies possible: use one program or the other. Clearly, the best way to find the optimal policy will be to simply try both of them in simulation, and see which one works better. There are also problems where pure policy search does not work well. One example is the following MDP:

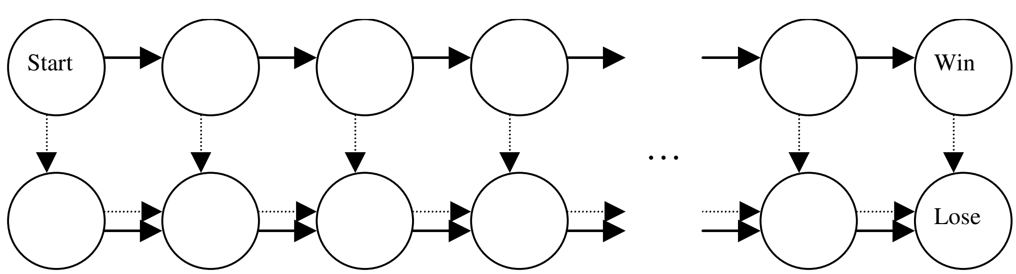

Figure 2.1.An MDP where pure policy search does poorly

In each state, the learning system must choose either the solid-line action or the dotted-line action. The only way to win is to choose the right action on every single time step. If the learning system must always start in the start state, and if the only reinforcement comes in the Win/Lose states, then it is very difficult to learn the policy directly. If there are N states in each row, then only one out of every 2^N policies will be optimal, and slight improvements to a suboptimal policy will never yield improvements in performance. If the learning system is allowed to choose which state to start in, then this can still be made difficult by adding an exponential number of new states that transition to the Win and Lose states, but aren't reachable from the Start state.

2.1.5 Dynamic Programming

For problems like figure 2.1, a better approach is to learn more than just a policy. For example, the learning system might remember which states are bad, with the rules:

1 The Lose state is bad.

2 If both arrows from a state lead to bad states, then that state is bad

3 If one arrow leaving a state goes to a bad state, then don't choose that action

Using this learning system, the agent can quickly learn to solve the problem. If it repeatedly starts in the Start state and performs random actions (except when rule 3 specifies an action), then it will naturally learn that the bottom states near the end are bad, and work its way back toward the beginning. After just O(N) runs, it will have a perfect policy.

Another approach would be to remember which arrows are bad rather than which states are bad. That could be done using these rules instead:

1 If an arrow goes to the Lose state, then that arrow is bad

2 If an arrow goes to a state with two bad arrows, then that arrow is bad

This also learns quickly. These two algorithms are known as incremental value iteration and Q-learning respectively. They are both forms of dynamic programming (Bertsekas, 1999). In general, dynamic programming algorithms learn a policy by storing more information than just the policy. They store *values*, which indicate how good states or state-action pairs are. Each value is updated according to the values of its successors. That causes information to flow back from end states toward the start state. Once the values have been learned, the policy becomes trivial: always choose the action that is greedy with respect to the learned values.

The two approaches to solving reinforcement-learning problems, pure policy search, or using values, tend to be used by different research communities, and are not generally combined. In chapter 6, it will be shown that through gradient-descent techniques, it is natural to combine the two approaches, and that in some cases the combination performs much better than either alone.

2.2 Reinforcement-Learning Algorithms

This section gives an overview of some of the reinforcement-learning algorithms in common use.

2.2.1 Actor-Critic

In actor-critic systems, there are two components to the reinforcement-learning system. The *critic* learns values, and the *actor* learns policies. At any given time, the critic is learning the values for the Markov chain that comes from following the current policy of the actor. The actor is constantly learning the policy that is greedy with the respect to the critic's current values.

It is particularly interesting to examine actor-critic systems that use a *lookup table* to store the values and policies. A lookup table represents the value in each state with a separate parameter. If it first updates the value in every state once, then updates the policy in every state once, then repeats, then this reduces to *incremental value iteration*, which is a form of dynamic programming that is guaranteed to converge to the optimal policy. If it instead updates all the values repeatedly in all the states until the values converge, then updates all the policies once, then repeats, then it reduces to *policy iteration*, another form of dynamic programming with guaranteed convergence. If it updates all the values N times between updating the policies, then it reduces to *modified policy iteration*, which is also guaranteed to converge to optimality.

It would seem that an actor-critic system with a lookup table is guaranteed to converge to optimality no matter what. Surprisingly, that is not the case. Although it always converges for $\gamma<0.5$ (Williams & Baird 1993), it does not always converge for larger γ, as shown by the following counterexample:

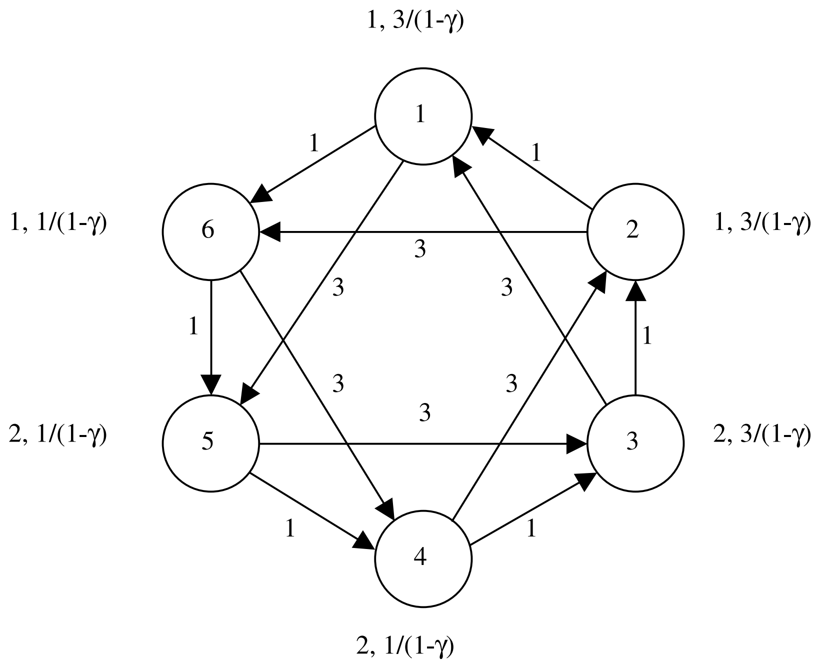

Figure 2.2. An MDP where actor-critic can fail to converge

The number on each arrow in figure 2.2 is the reinforcement. In each state, there is a choice between moving either 1 or 2 states around the circle. The first number at each state is the current policy (to move 1 or 2 states), and the second number is the current value (which is a function of the discount factor). Let B_i be the act of updating the value in state i to match the value of its successor under the current policy. Let I_i be the act of improving the policy in state i to be greedy with respect to the current values of the immediate successors. Performing the following updates in the order listed (reading from left to right) causes the policies and values to oscillate:

$$B_1, I_3, B_4, I_6, B_2, I_4, B_5, I_1, B_3, I_5, B_6, I_2$$

This updates every state's value and policy exactly once, yet leaves the policies in half the states being wrong. It can be repeated forever without every converging. If fact, even if the initial values are perturbed slightly, it will still oscillate forever.

On the other hand, randomly-selected B and I operations will converge with probability 1. This is obvious, since when there are optimal policies and values everywhere, no further changes are possible. There is a finite-length sequence of updates that will reach those optimal policies and values, and any finite sequence will be generated eventually with probability 1.

2.2.2 *Q*-learning

A more common algorithm is *Q*-learning. In *Q*-learning, a value $Q(x,u)$ is stored for each state x and action u. A Q value is updated according to:

$$Q(x_t,u_t) \longleftarrow (1-\alpha)Q(x_t,u_t) + \alpha \max_u \big(R_t + \gamma Q(x_{t+1},u)\big), \tag{2.1}$$

where a is a small, positive learning rate. A Q value $Q(x,u)$ is an estimate of the expected total discounted reinforcement received when starting in state x, performing action u, and following the optimal policy thereafter. The optimal policy is the policy that is greedy with respect to the optimal Q function. The optimal Q function is the unique function that satisfies this relationship between each Q value and the Q values of its successor state:

$$Q(x_t,u_t) = \max_u \big(E\big[R_t + \gamma Q(x_{t+1},u)\big]\big) \tag{2.2}$$

Equation (2.2) is the *Bellman equation* for *Q*-learning. The update in equation (2.1) can be thought of as changing the left side of the Bellman equation to more nearly match a sample of the right side. It must move slowly because the right side of (2.2) is an expected value, averaged over all possible successor states, while the right side of (2.1) is just a random sample of a successor state. *Q*-learning has guaranteed convergence with lookup tables if the learning rate decreases over time at an appropriate rate, and the Q values are stored in a lookup table (Watkins, 1989).

For a particular Q function, the difference between the two sides of equation (2.2) is the *Bellman residual*. Suppose that for a particular Q function, the worst Bellman residual for any state-action pair is an absolute difference of δ. Since this Q function is wrong, the policy that is greedy with respect to it may also be wrong. How bad can the greedy policy be? If the very first time step is in a state with a Bellman residual of δ, then the greedy policy might be suboptimal, transitioning to states whose expected max Q values are lower than for the optimal action by an amount of δ. In the long run, this may lower the total, expected, discounted reinforcement by at most δ. On the second time step, there might be anther error of at most δ, which lowers the total by at most $\gamma\delta$. In the long run, the total return may be too low by $\delta(1+\gamma+\gamma^2+\gamma^3+\ldots)=\delta/(1-\gamma)$. This kind of error bound is typical for reinforcement-learning algorithms based on dynamic programming. They are typically proportional to the maximum Bellman residual, and inversely proportional to $(1-\gamma)$. That is unfortunate when γ is close to 1, because that leads to a very large bound. Unfortunately, these bounds are tight: there are cases where the error really is that bad.

2.2.3 SARSA

In the algorithms discussed so far, it is assumed that states and actions are somehow chosen for training. It might be that they are chosen randomly, or it might be that they are chosen by following some trajectory generated by a random policy. One reasonable idea would be to start in the start state, and on every time step, choose the action that is greedy with probability 1-ε, and a random action with probability ε, for some small, positive ε. It might even be argued that the randomness should never be turned off, just in case the environment changes. If that is the case, then perhaps it would be better to learn the policy that is optimal, given that you will explore ε of the time. It would be like a person who when walking always takes a random step every 100 paces or so. Such a person would avoid walking along the top of a cliff, even when that is the "optimal" policy for a person who doesn't explore randomly.

SARSA is an algorithm that uses this idea. The update is:

$$Q(x_t,u_t) \longleftarrow (1-\alpha)Q(x_t,u_t) + \alpha\big(R_t + \gamma Q(x_{t+1},u_{t+1})\big)$$

This is the same as Q-learning, except that the value of the next state is not the maximum Q value. Instead, it is the Q value associated with whatever action is chosen at time t+1. That action will be the greedy action with probability 1-ε. In that case, the update is identical to Q-learning. With probability ε, the action will be random, and the value that is backed up will be lower.

3 Gradient Descent

This chapter describes the various forms of incremental and stochastic gradient descent, and the convergence results that have been proved. This will be the theoretical foundation for the algorithms proposed in chapters 4, 5, and 6.

3.1 Gradient Descent

Given a smooth, nonnegative, scalar function $f(\mathbf{x})$, how can the vector $\mathbf{x}$ that minimizes f be found? One approach is *gradient descent*. The vector $\mathbf{x}$ is initialized to some random value, and then on each time step, it is updated according to:

$$\mathbf{x} \longleftarrow \mathbf{x} - \alpha \nabla_{\mathbf{x}} f(\mathbf{x})$$

Clearly, $f(\mathbf{x})$ will tend to decrease over time. It may eventually get near a local minimum, and then start to oscillate as it bounces back and forth across the bottom. To get $f(\mathbf{x})$ to converge, it is usually necessary to shrink the learning rate over time, so the oscillations will decrease. If the learning rate shrinks too fast, though, $\mathbf{x}$ may converge to a point that isn't a local minimum. The standard conditions on the learning rate are that it shrinks according to some schedule such that the following two conditions hold:

$$\sum_{t=0}^{\infty} \alpha_t = \infty \tag{3.1}$$

$$\sum_{t=0}^{\infty} \alpha_t^2 < \infty \tag{3.2}$$

Simple gradient-descent methods are almost never used with reinforcement learning, supervised learning, or any of the problems or algorithms mentioned in this thesis. Instead, it is much more common to use incremental gradient descent, stochastic gradient descent, or both.

3.2 Incremental Gradient Descent

The previous section assumed that $f(\mathbf{x})$ was an arbitrary, smooth function. Suppose, instead, that $f(\mathbf{x})$ is defined to be the sum of a large number of individual functions:

$$f(\mathbf{x}) = \sum_{i=1}^{n} f_i(\mathbf{x})$$

Given this definition of $f(\mathbf{x})$, simple gradient descent would be to repeatedly change $\mathbf{x}$ according to:

$$\mathbf{x} \longleftarrow \mathbf{x} - \alpha \sum_{i=1}^{n} \nabla_{\mathbf{x}} f_i(\mathbf{x})$$

Incremental gradient descent repeats this instead:

$$\begin{aligned}&\text{for } i = 1 \text{ to } n\\ &\quad \mathbf{x} \longleftarrow \mathbf{x} - \alpha \nabla_{\mathbf{x}} f_i(\mathbf{x})\end{aligned}$$

This is often used, for example, with backpropagation neural networks. In that case $\mathbf{x}$ is a weight vector for the neural network, and $f_i(\mathbf{x})$ is the squared error in the output for training example *i*. Then $f(\mathbf{x})$ is the total squared error. Simple gradient descent corresponds to epoch-wise training, and incremental gradient descent corresponds to incremental training, where the weights are changed immediately after each training example is presented.

3.3 Stochastic Gradient Descent

Incremental training assumes that each of the $f_i(\mathbf{x})$ functions are evaluated in turn before starting over on the first one. Alternatively, one could just pick the $f_i(\mathbf{x})$ functions randomly from the set by repeatedly doing:

$$\begin{aligned}&i \longleftarrow \text{random number in } [1, n]\\ &\mathbf{x} \longleftarrow \mathbf{x} - \alpha \nabla_{\mathbf{x}} f_i(\mathbf{x})\end{aligned}$$

This is stochastic gradient descent. On each time step, the $\mathbf{x}$ vector changes by a random amount, but on average it is moving in the direction of the gradient. As the learning rate shrinks, these small steps start to average out, and it is very much like doing simple gradient descent.

3.4 Unbiased Estimators

Of course, there are other forms of stochastic gradient descent as well. The most general form is to repeatedly do:

$$\mathbf{x} \longleftarrow \mathbf{x} - \alpha \left(\nabla_{\mathbf{x}} f(\mathbf{x}) + \mathbf{w} \right) \tag{3.3}$$

where $\mathbf{w}$ is a random, zero-mean vector chosen independently each time from some fixed probability distribution. The stochastic gradient descent in the previous section is just a special case of update (3.3). The expression in parentheses in (3.3) is correct on average, so its value on any given time step is an *unbiased estimate* of the true gradient. Let **Y** and **Z** be random variables, and let y_1, y_2, z_1, and z_2 be samples from those random variables. If $E[]$ is the expected value, then it is the case that:

y_1 is an unbiased estimate of $E[\mathbf{Y}]$

∇y_1 is an unbiased estimate of $E[\nabla\mathbf{Y}]$

y_1+z$_1$ is an unbiased estimate of $E[\mathbf{Y+Z}]$

$y_1\, z_1$ is an unbiased estimate of $E[\mathbf{Y}\ \mathbf{Z}]$

$y_1\, y_2$ is an unbiased estimate of $E[\mathbf{Y}^2]$

$2\, y_1\, \nabla y_2$ is an unbiased estimate of $\nabla E[\mathbf{Y}^2] = E[\nabla(\mathbf{Y}^2)]$

$2\, y_1\, \nabla y_1$ is an unbiased estimate of $(\nabla E[\mathbf{Y}])^2 = (E[\nabla\mathbf{Y}])^2$

The last two lines are particularly important. True stochastic gradient descent requires unbiased estimates of the gradient. To get the expected gradient of the square of a random variable requires two independent samples (y_1 and y_2). If the same sample is used twice, this does yield an unbiased estimate of *something*, but it's not the expected value of a square any more. This is significant for most of the algorithms proposed in this thesis. Convergence to a local minimum of the mean squared Bellman residual is guaranteed using two independent samples. If a single sample is used twice, then it minimizes the *squared expected value* rather than the *expected squared value*. Depending on how random the MDP is, this might cause the policy to be fairly suboptimal.

3.5 Known Results for Error Backpropagation

A large literature exists for backpropagation convergence results, based on the general literature for stochastic approximation. The convergence of stochastic and incremental algorithms for neural networks has been extensively studied (White 1989, White 1990, Gaivoronski 1994, Mangasarian & Solodov 1994, Luo & Tseng 1994, Solodov 1995, Mangasarian & Solodov 1995, Solodov 1996, Luo 1991, Bertsekas 1995, Bertsekas & Tsitsiklis 1996, Solodov 1997, Solodov and Zavriev 1998). Over the last few years, results have been extended and generalized. Two of the latest papers are most relevant to the algorithms in this thesis.

If the $f(\mathbf{x})$ function is smooth and has a Lipshitz continuous gradient, then a huge range of results can be proved (Bertsekas & Tsitsiklis, 1997, revised Jan 1999). If f is nonnegative and the learning rate decays according to equations (3.1) and (3.2), then $f(\mathbf{x})$ will converge, its gradient will converge to zero, and every limit point of $\mathbf{x}$ is a stationary point of f, all with probability 1. In other words, it is guaranteed to converge to a local minimum in every desirable sense. In fact, for the incremental version (rather than stochastic), the convergence is absolutely guaranteed, rather than just with probability 1. Most function approximators satisfy the smoothness assumptions, so any simple error function like mean squared error will also satisfy them.

Even the smoothness assumptions can be relaxed, allowing piecewise smooth functions that contain creases where the gradient doesn't even exist (Solodov 1995). These results apply to incremental gradient descent. It is interesting to ask what the definition of "local minimum" will be when there are creases. Obviously, it can't be that the gradient will converge to zero, since that can't happen when doing gradient descent on a function like $f(x)=|x|$. The corresponding concept for nonsmooth functions is that **x** converges to a point whose generalized derivative includes the zero vector. In other words, **x** converges to a local minimum, even if the gradient isn't defined at that point. The full result is that f will converge certainly, and **x** will converge to a local minimum if **x** remains bounded (which in turn is assured if a weight decay term is added).

Each of the algorithms in this thesis is said to converge "in the same sense as backpropagation". This means that if they are executed with incremental gradient descent (such as during prioritized sweeping), then convergence is guaranteed by the Solodov results in every sense that would be wanted. If the algorithms are executed with smooth error functions, then the Bertsekis and Tsitsiklis results guarantee convergence in every sense that would be wanted. In fact, these results are even stronger than are needed.

That still leaves one other case. What if it is desired to do stochastic gradient descent rather than incremental (e.g. during reinforcement learning with random exploration), and the error function is not smooth? Reinforcement learning differs from Backpropagation in that this case of nonsmooth error functions can actually occur, even when the function approximator appears at first glance to be very smooth. The problem arises because of the *max* operator. If a neural network is infinitely differentiable and has two outputs, $Q_1(\mathbf{x})$ and $Q_2(\mathbf{x})$, corresponding to two different actions, then the value function is defined as:

$$V(\mathbf{x},\mathbf{w}) = \max(Q_1(\mathbf{x},\mathbf{w}), Q_2(\mathbf{x},\mathbf{w}))$$

In this case, even if Q_1 and Q_2 are smooth functions of the weights, V probably isn't. This can be seen by considering the case when $Q_1(\mathbf{x},\mathbf{w})=Q_2(\mathbf{x},\mathbf{w})$. Suppose that a infinitessimal increase in a given weight causes Q_1 to increase but not Q_2. Then a small increase in that weight will cause V to increase, but a small decrease in that weight will not change V at all. That means that the derivative of V with respect to that weight will not exist at that point. Most of the algorithms proposed here have error functions that are functions of a *max*, so this would make the error functions nonsmooth. Even worse, there is no way to fix the problem by using some kind of soft max function. In dynamic programming, the maximum is a very important function. Any smoothing of it would introduce errors, and even a small error introduced on every time step can lead to a large error in the final policy.

So is it hopeless? Not at all. It turns out the function approximator wasn't as smooth as it initially looked, but it can easily be made smooth without changing it much at all. The solution is to call the outputs of the function approximator y_1 and y_2 instead of Q_1 and Q_2. Then, a simple function calculates Q_1 and Q_2 as a function of y_1 and y_2. This is done in such a way that Q_i is almost identical to y_i, Q_i is a smooth function of the weights, and the maximum of all the Q_i is itself a smooth function of the weights. The process doesn't even change the policy; the maximum Q_i will be the same i as the maximum y_i.

One possible example of such a smoothing function is given here. It will ensure that all of the derivatives are continuous. It could be much simpler if it just ensure that the first and second derivatives were continuous.

First, define each Q value to be a weighted average of the y values, as shown in equation (3.4).

$$Q_i(\mathbf{x},\mathbf{w}) = \frac{\sum_j g\big(y_j(\mathbf{x},\mathbf{w}) - y_i(\mathbf{x},\mathbf{w})\big) y_j(\mathbf{x},\mathbf{w})}{\sum_j g\big(y_j(\mathbf{x},\mathbf{w}) - y_i(\mathbf{x},\mathbf{w})\big)} \tag{3.4}$$

where g is a smooth, positive function that approaches zero for large positive and negative arguments. In other words, each Q will be a weighted average of all the y values, but it will give the most weight to its own y value and y values close to its own, and very little weight to y values that are much different from its own. One possible choice for the g function is equation (3.5), which is graphed in figure 3.1.

$$g(x) = \begin{cases} e^{\frac{x}{\varepsilon}\left(1-0.3e^{\frac{7}{3}\left(1-\frac{\varepsilon}{x}\right)}\right)} & \text{if } x > 0 \\ e^{\frac{x}{\varepsilon}} & \text{otherwise} \end{cases} \tag{3.5}$$

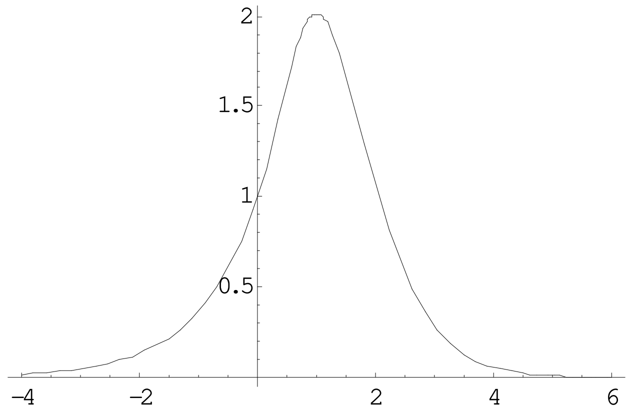

Figure 3.1. The g function used for smoothing. Shown with ε=1.

This filter on the output of the function approximator causes the V function to be smooth, yet has a very small effect on the nature of the Q function. Its properties include:

- The *policy* is unchanged. The maximum Q corresponds to the maximum y. Q values will be tied for maximum if and only if the corresponding y values are tied.

- The values change little. If all the y values are spread out, with no two being close, then each Q will be almost equal to the corresponding y. If two or more y values are close to each other, then the corresponding Q values will be drawn closer to the mean of those y values. In either case, the Q value is close to the corresponding y value. In each case, the meaning of "close" is controlled by ε. For any given set of y values, as ε goes to zero, each Q value goes to the corresponding y value.

- It is computationally cheap. Very little calculation is needed to find Q from y, especially compared to the computation needed to find y when y is a neural network. Furthermore, some algorithms, such as VAPS (chapter 6) with $\phi>0$, or Wire Fitting (Baird & Klopf, 1993), already pass the output of y through a similar-looking function, so there is very little additional cost to fold in this new calculation.

- It makes *max* smooth. The partial derivative of V with respect to each y exists everywhere. The partial derivative of each Q with respect to each y or weight also exists. The second and higher derivatives can also be made to exist if desired.

Each of these properties is easily shown. Since g is a function of x/ε, reducing ε will cause the peak to become narrower, causing each y to have less effect on other y values that are far from it. Clearly, as ε goes to zero, every Q will therefore approach the corresponding y.

To show that smoothing does not affect the policy (the largest y corresponds to the largest Q), first consider what would happen if g were exponential for all x, rather than just for $x \leq 0$. In that case, plugging the exponential in for g in equation (3.4) causes the y_i terms in the numerator and denominator of equation (3.4) to cancel, leaving an expression that does not depend on y_i. So, if g were a simple exponential, all of the Q values would be equal to each other. Next, consider what happens when the g defined in equation (3.5) is used instead. Note that when x is positive, $g(x)<e^{x/\varepsilon}$. This must be true because $g(x)$ is defined in that case to be of the form $g(x)=e^{x/\varepsilon(1-f(x))}$, where $f(x)$ is a positive expression, which makes g less than the simple exponential. Note that for the maximum y_i, all of the differences are negative, so the Q_i value will be the same for the simple exponential as for the g from equation (3.4). For any y value that is not the maximum, the weight that it gives to y values greater than itself is decreased when g is changed from a simple exponential to equation (3.4). Since it gives reduced weight to y values greater than itself, and the same weight to itself and values less than itself, its Q value will decrease. So, for the simple exponential, all Q values are the same, and then changing g to use equation (3.4) causes the Q values associated with the maximum y to stay the same, and all others to decrease. Therefore, the smoothing preserves policies. All of this only works because $g(x)<e^{x/\varepsilon}$, and that is why g was specifically chosen to have that property.

It is also easy to show that g is continuous, as is its first derivative with respect to the y values (the gradient), its second derivative (the Hessian), and all higher derivatives. Clearly this will be true at points other than $g(0)$. At $g(0^-)$, the simple exponential has a value of $e^{x/\varepsilon}$, and an nth derivative of $\varepsilon^{-n}e^{x/\varepsilon}$. For $x>0$, g is of the form $g(x)=e^{x/\varepsilon(1-f(x))}$, and it is clear that $f(x)$ is an expression whose value and all derivatives at 0 go to zero when approached from the right. Given that fact, it is clear that $g(0^+)$ itself must have the appropriate derivatives when approached from the right. The derivative of the right half of $g(x)$ will be the sum of two terms: $\varepsilon^{-1}g(x)-f\,'(x)g(x)$. The second term contains an f, which makes it zero at $x=0^+$, and makes all further derivatives of it zero there. So the second term can be ignored when taking further derivatives. The first term is the same as when taking the derivative of the simple exponential. Further derivatives follow the same pattern. Therefore, $g(0^-)=g(0^+)$ and also $g\,'(0^-)=g\,'(0^+)$ and $g\,''(0^-)=g\,''(0^+)$ and so on.

Finally, this smoothing function makes the maximum operator smooth. This is obviously true when there is a unique maximum y. To consider the case of a tie, plug the definition of g into the definition of Q, and take the derivatives for the maximum Q value with respect to all the y values. Note that for the maximum Q, every g behaves just like a simple exponential. Taking the derivative of the combined equation, and looking at the limit as the second-largest y approaches the largest y, it is clear that the gradients of each of them with respect to all the y values (including each other) are equal. This only works because g is a simple exponential for $x<0$. That is why g was specifically chosen to have that property.

When the function approximator is smoothed in this way, the algorithms discussed in this thesis converge to a local minimum in the same sense as backpropagation. It is interesting that convergence proofs for supervised learning require smooth function approximators, and now convergence proofs for reinforcement learning also require smooth function approximators. However, in the reinforcement learning case, the smoothness constraint deals with the derivative of the maximum output, not just the derivatives of each output individually. As in supervised learning, it is not difficult to ensure function approximators have the needed property. In fact, as shown in this section, any function approximator that is smooth in the supervised-learning sense can be made smooth in the reinforcement-learning sense with a small modification. This modification has little effect on the Q values, little effect on the computational cost, and no effect on the policy. Neither this nor decaying learning rates were needed for any of the simulations in this thesis.

4 Residual Algorithms: Guaranteed Convergence with Function Approximators

Reinforcement learning is often done using function approximators. Although there is a well-developed theory guaranteeing reinforcement-learning convergence on lookup tables, and although there is a well-developed theory guaranteeing supervised-learning convergence on function approximators, little has been proved about the combination of the two. This chapter demonstrates that when the two concepts are combined in the obvious way, as has normally been done, the algorithms can diverge. This chapter shows very simple problems where these algorithms blow up, proposes *residual gradient* algorithms, which have provable convergence, and proposes *residual algorithms*, which maintain the guarantees while learning faster in practice.

4.1 Introduction

A number of reinforcement learning algorithms have been proposed that are guaranteed to learn a *policy*, a mapping from states to actions, such that performing those actions in those states maximizes the expected, total, discounted reinforcement received:

$$V = \left\langle \sum_t \gamma^t R_t \right\rangle \tag{4.1}$$

where R_t is the reinforcement received at time t, <> is the expected value over all stochastic state transitions, and γ is the discount factor, a constant between zero and one that gives more weight to near-term reinforcement, and that guarantees the sum will be finite for bounded reinforcement. In general, these reinforcement learning systems have been analyzed for the case of an MDP with a finite number of states and actions, and for a learning system containing a lookup table, with separate entries for each state or state-action pair. Lookup tables typically do not scale well for high-dimensional MDPs with a continuum of states and actions (the curse of dimensionality), so a general function-approximation system must typically be used, such as a sigmoidal, multi-layer perceptron, a radial-basis-function network, or a memory-based-learning system. In the following sections, various methods are analyzed that combine reinforcement learning algorithms with function approximation systems. Algorithms such as Q-learning or value iteration are guaranteed to converge to the optimal answer when used with a lookup table. The obvious method for combining them with function-approximation systems, called the *direct* algorithm here, does not have those guarantees. If fact, counterexamples will be shown that demonstrate both direct Q-learning and direct value iteration failing to converge to an answer. Even batch training and *on-policy* training doesn't help direct Q-learning in that example. A new class of algorithms, *residual gradient* algorithms, are shown to always converge, but residual gradient Q-learning and residual gradient value iteration may converge very slowly in some cases. Finally, a new class of algorithms, *residual* algorithms, are proposed. It will be shown that direct and residual gradient algorithms are actually special cases of residual algorithms, and that residual algorithms can easily be found such that residual Q-learning or residual value iteration have both guaranteed convergence, and converge quickly on problems for which residual gradient algorithms converge slowly. This chapter does not just define a new algorithm. Rather, it defines a new process for deriving algorithms from first principles. Using this process, the residual form of any reinforcement learning algorithm based on dynamic programming can be easily derived. This new algorithm is then guaranteed to converge, and may even learn faster in practice, which is shown in simulation here. In addition, this framework will form the basis of the algorithms proposed in chapters 5 and 6, which are also types of residual algorithms.

4.2 Direct Algorithms

If a Markov chain has a finite number of states, and each $V(x)$ is represented by a unique entry in a lookup table, and each possible transition is experienced an infinite number of times during learning, then update **Error! Reference source not found.** is guaranteed to converge to the optimal value function as the learning rate α decays to zero at an appropriate rate. The various states can be visited in any order during learning, and some can be visited more often than others, yet the algorithm will still converge if the learning rates decay appropriately (Watkins, Dayan 92). If $V(x)$ was represented by a function-approximation system other than a lookup table, update **Error! Reference source not found.** could be implemented directly by combining it with the backpropagation algorithm (Rumelhart, Hinton, Williams 86). For an input x, the actual output of the function-approximation system would be $V(x)$, the "desired output" used for training would be $R+\gamma V(x')$, and all of the weights would be adjusted through gradient descent to make the actual output closer to the desired output. For any particular weight w in the function-approximation system, the weight change would be:

$$\Delta w = \alpha\big(R + \gamma V(x_{t+1}) - V(x_t)\big)\frac{\partial V(x_t)}{\partial w} \tag{4.2}$$

Equation (4.2) is exactly the TD(0) algorithm, by definition. It could also be called the *direct* implementation of incremental value iteration or Q-learning. The direct algorithm reduces to the original algorithm when used with a lookup table. Tesauro (1990, 1992) has shown very good results by combining TD(0) with backpropagation (and also using the more general TD(λ)). Since it is guaranteed to converge for the lookup table, this approach might be expected to also converge for general function-approximation systems. Unfortunately, this is not the case, as is illustrated by the tiny MDP shown in figure 4.1.

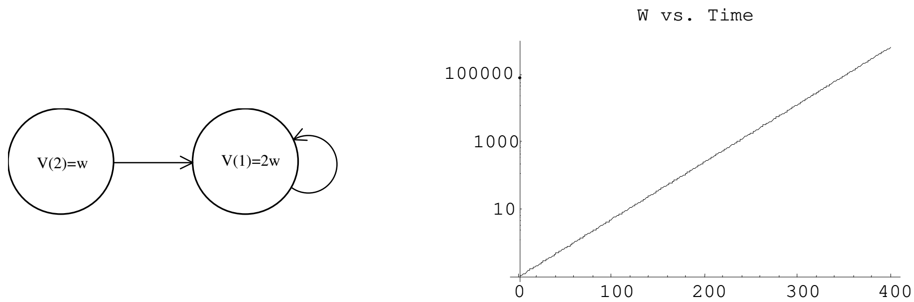

Figure 4.1. The 2-state problem for value iteration, and a plot of the weight vs. time. R=0 everywhere and γ=0.9. The weight starts at 0.1, and grows exponentially, even with batch training, and even with arbitrarily-small learning rates.

In figure 4.1, the entire MDP is just two states, and the function approximator is linear, with only a single weight. There is zero reinforcement on each time step, and the discount factor γ=0.9. The optimal weight is zero, giving correct values of zero in each state. Unfortunately, if the initial weight is nonzero, then it will grow without bound, and the values will grow without bound. This problem happens whether training is batch or incremental, and no matter what positive learning rate is chosen, even a slowly-decreasing learning rate. It is disturbing that a widely-used algorithm would fail on such a simple problem.

This MDP has no absorbing state. Trajectories go forever. Could it be that MDPs with finite-length trajectories will always avoid the problem seen here? No. Any MDP with a discount factor of γ can be transformed into a new MDP with no discounting (a discount factor of 1.0), with a new absorbing state added, and with a transition from every other state to the absorbing state with probability 1-γ. If that transformation is done to any of the counterexamples given in this thesis, the weights will still change in exactly the same way, and the values will change in exactly the same way. So whether trajectories eventually end with probability 1 or just go on forever, either way the counterexamples blow up the same way.

Could the problem be that the function approximator is not general enough? After all, it is able to represent the optimal value function, $v(0)=v(1)=0$, but there exist other value functions that it cannot represent, such as $v(0)=v(1)=1$. No, even that does not prevent divergence in general, as shown by figure 4.2.

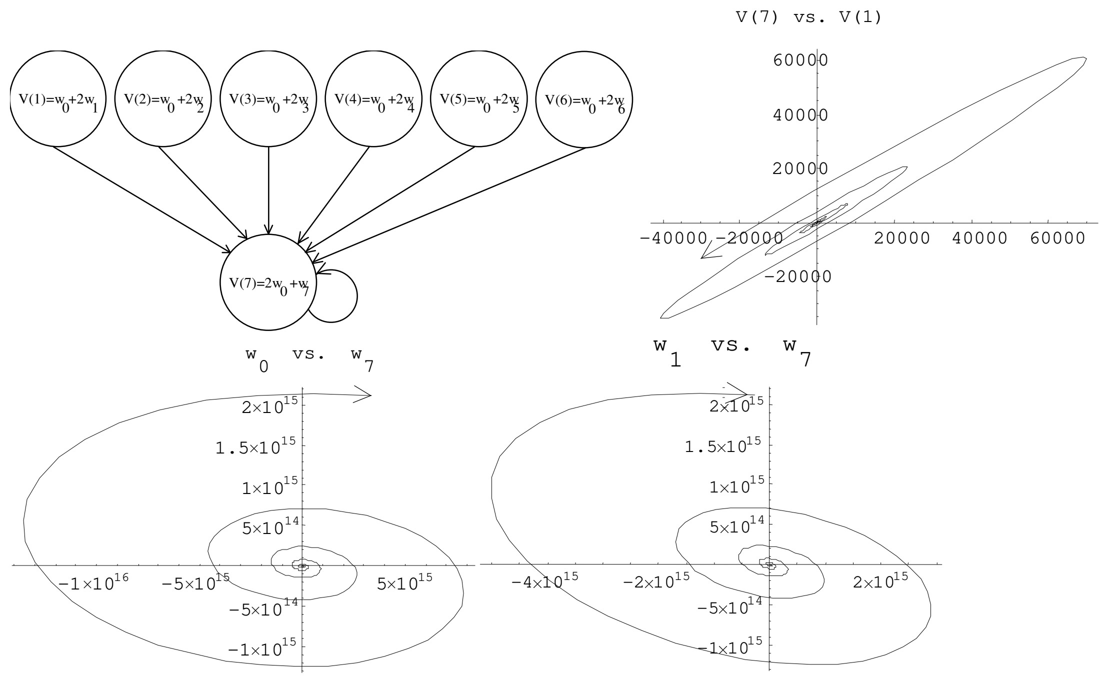

Figure 4.2. The 7-state star problem for value iteration, and a plot of the values and weights spiraling out to infinity, where all weights started at 0.1. By symmetry, weights 1 through 6 are always identical. R=0 everywhere and γ=0.9.

In figure 4.2, there are seven states, and the value of each state is given by the linear combination of two weights. Every transition yields a reinforcement of zero. During training, each possible transition is observed equally often. The function-approximation system is simply a lookup table, with one additional weight giving generalization. This is an extremely benign form of function-approximation system. It is linear, it is general (can represent any value function over those states), the state vectors are linearly independent, and all have the same magnitude (1-norm, 2-norm, or infinity-norm). Furthermore, it has the desirable property that using backpropagation to change the value in one state will cause neighboring states to change by at most two-thirds as much. Therefore, this system exhibits only mild generalization. If one wished to extend the Watkins and Dayan proofs to function-approximation systems, this would appear to be an ideal system for which convergence to optimality could be guaranteed for the direct method. However, that is not the case.

If the weight w_0 were not being used, then it would be a lookup table, and the weights and values would all converge to zero, which is the correct answer. However, in this example, if all weights are initially positive, then all of the values will grow without bound. This is due to the fact that when the first six values are lower than the value of their successor, $\gamma V(7)$, and $V(7)$ is higher than the value of its successor, $\gamma V(7)$, then w_0 is increased five times for every time that it is decreased, so it will rise rapidly. Of course, w_7 will fall, but more slowly, because it is updated less frequently. The net effect then is that all of the values and all of the weights grow without bound, spiraling out to infinity. It is also possible to modify the counterexample so the weights grow monotonically, rather than spiraling out. Figure 4.3 shows one such Markov chain. Note that the value of state 11 is always greater than that of state 1. This means that if the Markov chain were converted to an MDP, adding a choice of which state to go to, it might be expected to learn a policy that chooses to go to state 1.

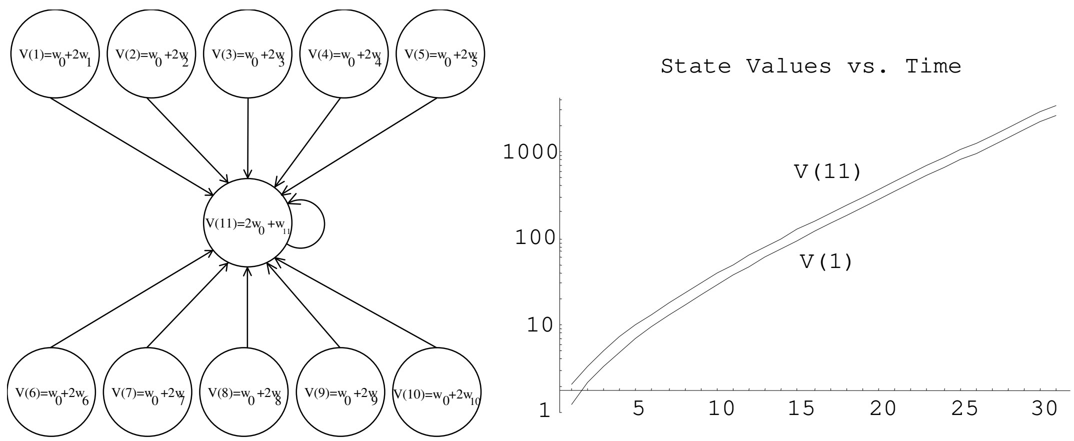

Figure 4.3. The 11-state star problem for value iteration, where all weights started at 0.1 except w_0, which started at 1.0. R=0 everywhere and γ=0.9.

In this example, every transition was updated equally often, even though the transition from state 1 to itself would be seen more often during an actual trajectory. What if training were limited to *on-policy* learning? This is learning where the states are updated with frequencies proportional to how often they are seen during trajectories, while following a single, fixed policy. On-policy learning includes learning on states as they are seen on a trajectory, or learning on randomly-chosen states from a database of states gleaned from trajectories. On-policy training does guarantee convergence for linear function approximators when the problem is purely prediction on a Markov chain (there are no actions or decisions). Could this proof be extended to Q-learning on MDPs? No, even with on-policy training and general, linear function approximators, Q-learning can still blow up, as demonstrated in figure 4.4.

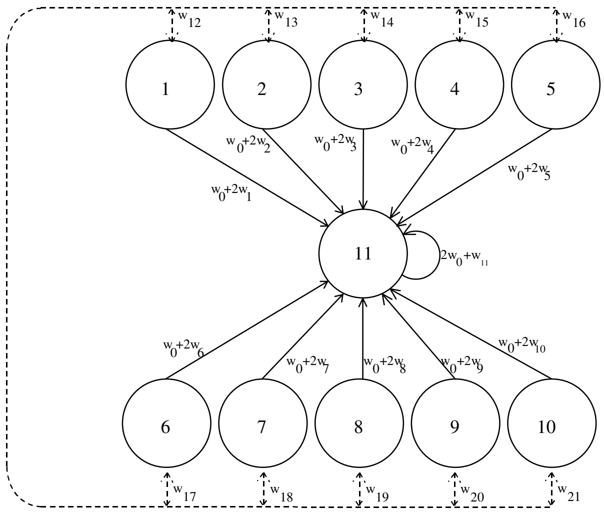

Figure 4.4.The star problem for *Q*-learning. *R*=0 everywhere and γ=0.9.

In this MDP, every transition receives zero reinforcement, and each state has two actions, one represented by a solid line, and one represented by a dotted line. In all states, the solid action transitions to state 11, and the dotted action transitions to one of the states 1 through 10, chosen randomly with uniform probability. During training, a fixed stochastic policy is used to ensure sufficient exploration. In every state, the solid action is chosen with probability 1/10, and the dotted action is chosen with probability 9/10. This ensures that every state-action pair is explored infinitely often, and that each of the solid *Q* values is updated equally often. If the solid *Q* values start larger than the dotted *Q* values, and the transition from state 11 to itself starts out as the largest of the solid *Q* values, then all weights, *Q* values, and values will diverge to infinity. As long as the policy in every state is to do the solid action, the solid *Q* values will act just like the state values in the example in figure 4.3. That ensures that the state values will all blow up monotonically, which in turn ensures that the policy will never change.

This is true for both epoch-wise and incremental learning, and even for small learning rates or slowly decreasing learning rates. This example demonstrates that for a simple MDP with a linear function approximator able to represent all possible *Q*-functions, the *Q* values can diverge, even when training on trajectories. The next section presents a way to modify *Q*-learning to ensure convergence to a local optimum.

4.3 Residual Gradient Algorithms

It is unfortunate that a reinforcement learning algorithm can be guaranteed to converge for lookup tables, yet be unstable for function-approximation systems that have even a small amount of generalization. Algorithms have been proved to converge for LQR problems with quadratic function-approximation systems (Bradtke 93), but it would be useful to find an algorithm that converges for any function-approximation system on more general problems. To find an algorithm that is more stable than the direct algorithm, it is useful to specify the exact goal for the learning system. For the problem of prediction on a deterministic Markov chain, the goal can be stated as finding a value function such that, for any state x and its successor state x', with a transition yielding immediate reinforcement R, the value function will satisfy the Bellman equation:

$$V(x) = \langle R + \gamma V(x') \rangle \tag{4.3}$$

where < > is the expected value over all possible successor states x'. For a system with a finite number of states, the optimal value function V^* is the unique function that satisfies the Bellman equation. For a given value function V, and a given state x, the *Bellman residual* is defined to be the difference between the two sides of the Bellman equation. The *mean squared Bellman residual* for an MDP with n states is therefore defined to be:

$$E = \frac{1}{n} \sum_x \langle R + \gamma V(x') - V(x) \rangle^2 \tag{4.4}$$

If the Bellman residual is nonzero, then the resulting policy will be suboptimal, but for a given level of Bellman residual, the degree to which the policy yields suboptimal reinforcement can be bounded (section 2.2.2, and Williams, Baird 93). This suggests it might be reasonable to change the weights in the function-approximation system by performing stochastic gradient descent on the mean squared Bellman residual, E. This could be called the *residual gradient* algorithm. Residual gradient algorithms can be derived for both Markov chains and MDPs, with either stochastic or deterministic systems. For simplicity, it will first be derived here for a deterministic Markov chain, then extended in the next section. Assume that V is parameterized by a set of weights. To learn for a deterministic system, after a transition from a state x to a state x', with reinforcement R, a weight w would change according to:

$$\Delta w = -\alpha \left[R + \gamma V(x') - V(x) \right] \left[\tfrac{\partial}{\partial w} \gamma V(x') - \tfrac{\partial}{\partial w} V(x) \right] \tag{4.5}$$

For a system with a finite number of states, E is zero if and only if the value function is optimal.

In addition, because these algorithms are based on gradient descent, it is trivial to combine them with any other gradient-descent-based algorithm, and still have guaranteed convergence. For example, they can be combined with weight decay by adding a mean-squared-weight term to the error function. My Ph.D. student, Scott Weaver, developed a gradient-descent algorithm for making neural networks become more local automatically (Weaver, 1999). This could be combined with residual gradient algorithms by simply adding his error function to the mean squared Bellman residual. The result would still have guaranteed convergence.

Although residual gradient algorithms have guaranteed convergence, that does not necessarily mean that they will always learn as quickly as direct algorithms, nor that they will find as good a final solution. Applying the direct algorithm to the example in figure 4.5 causes state 5 to quickly converge to zero. State 4 then quickly converges to zero, then state 3, and so on. Information flows purely from later states to earlier states, so the initial value of w_4, and its behavior over time, has no effect on the speed at which $V(5)$ converges to zero. Applying the residual gradient algorithm to figure 4.2 results in much slower learning. For example, if initially w_5=0 and w_4=10, then when learning from the transition from state 4 to state 5, the direct algorithm would simply decrease w_4, but the residual gradient algorithm would both decrease w_4 and increase w_5. Thus the residual gradient algorithm causes information to flow both ways, with information flowing in the wrong direction moving slower than information flowing in the right direction by a factor of γ. If γ is close to 1.0, then it would be expected that residual gradient algorithms would learn very slowly on the problem in figure 4.5.

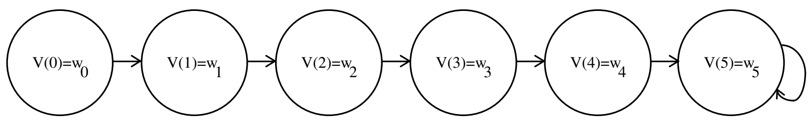

Figure 4.5. The hall problem. R=1 in the absorbing state, and zero everywhere else. γ=0.9.

4.4 Residual Algorithms

Direct algorithms can be fast but unstable, and residual gradient algorithms can be stable but slow. Direct algorithms attempt to make each state match its successors, but ignore the effects of generalization during learning. Residual gradient algorithms take into account the effects of generalization, but attempt to make each state match both its successors and its predecessors. These effects can be seen more easily by considering *epoch-wise* training, where a weight change is calculated for every possible state-action pair, according to some distribution, then the weight changes are summed and the weights are changed appropriately. In this case, the total weight change after one epoch for the direct method and the residual gradient method, respectively, are:

$$\Delta \mathbf{W}_d = -\alpha \sum_x \left[R + \gamma V(x') - V(x)\right]\left[-\nabla_{\mathbf{W}} V(x)\right] \tag{4.6}$$

$$\Delta \mathbf{W}_{rg} = -\alpha \sum_x \left[R + \gamma V(x') - V(x)\right] \left[\nabla_{\mathbf{W}} \gamma V(x') - \nabla_{\mathbf{W}} V(x)\right] \tag{4.7}$$

In these equations, $\mathbf{W}$, $\Delta\mathbf{W}$, and the gradients of $V(x)$ and $V(x')$ are all vectors, and the summation is over all states that are updated. If some states are updated more than once per epoch, then the summation should include those states more than once. The advantages of each algorithm can then be seen graphically.

Figure 4.6 shows a situation in which the direct method will cause the residual to decrease (left) and one in which it causes the residual to increase (right). The latter is a case in which the direct method may not converge. The residual gradient vector shows the direction of steepest descent on the mean squared Bellman residual. The dotted line represents the hyperplane that is perpendicular to the gradient. Any weight change vector that lies to the left of the dotted line will result in a decrease in the mean squared Bellman residual, *E*. Any vector lying along the dotted line results in no change, and any vector to the right of the dotted line results in an increase in *E*. If an algorithm always decreases *E*, then clearly *E* must converge. If an algorithm sometimes increases *E*, then it becomes more difficult to predict whether it will converge. A reasonable approach, therefore, might be to change the weights according to a weight-change vector that is as close as possible to $\Delta\mathbf{W}_d$, so as to learn quickly, while still remaining to the left of the dotted line, so as to remain stable. Figure 4.7 shows such a vector.

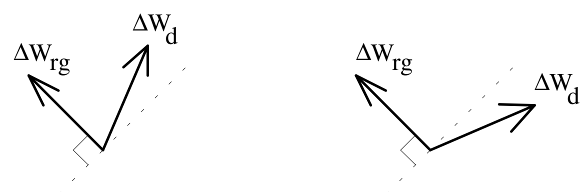

Figure 4.6. Epoch-wise weight-change vectors for direct and residual gradient algorithms

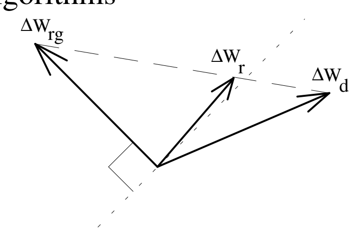

Figure 4.7. Weight-change vectors for direct, residual gradient, and residual algorithms.

This weighted average of a direct algorithm with a residual gradient algorithm could have guaranteed convergence, because $\Delta\mathbf{W}_r$ causes *E* to decrease, and might be expected to be fast, because $\Delta\mathbf{W}_r$ lies as close as possible to $\Delta\mathbf{W}_d$: Actually, the closest stable vector to $\Delta\mathbf{W}_d$ could be found by projecting $\Delta\mathbf{W}_d$ onto the plane perpendicular to $\Delta\mathbf{W}_{rg}$, which is represented by the dotted line. However, the resulting vector would be collinear with $\Delta\mathbf{W}_r$, so $\Delta\mathbf{W}_r$ should learn just as quickly for appropriate choices of learning rate. $\Delta\mathbf{W}_r$ is simpler to calculate, and so appears to be the most useful algorithm to use. For a real number ϕ between 0 and 1, $\Delta\mathbf{W}_r$ is defined to be:

$$\Delta\mathbf{W}_r = (1-\phi)\Delta\mathbf{W}_d + \phi\Delta\mathbf{W}_{rg} \tag{4.8}$$

This algorithm is guaranteed to converge for an appropriate choice of ϕ. The algorithm causes the mean squared residual to decrease monotonically (for appropriate ϕ), but it does not follow the negative gradient, which would be the path of steepest descent. Therefore, it would be reasonable to refer to the algorithm as a *residual* algorithm, rather than as a *residual gradient* algorithm. A residual algorithm is defined to be any algorithm in the form of equation (4.8), where the weight change is the weighted average of a residual gradient weight change and a direct weight change. By this definition, both direct algorithms and residual gradient algorithms are special cases of residual algorithms.

An important question is how to choose ϕ appropriately. One approach is to treat it as a constant that is chosen manually by trial and error, as is done when people use backpropagation with a constant learning rate. Just as a learning rate constant can be chosen to be as high as possible without causing the weights to blow up, so ϕ can be chosen as close to 0 as possible without the weights blowing up. A ϕ of 1 is guaranteed to converge, and a ϕ of 0 might be expected to learn quickly if it can learn at all. However, this may not be the best approach. It requires an additional parameter to be chosen by trial and error, and it ignores the fact that the best ϕ to use initially might not be the best ϕ to use later, after the system has learned for some time.

Fortunately, it is easy to calculate the ϕ that ensures a decreasing mean squared residual, while bringing the weight change vector as close to the direct algorithm as possible. To accomplish this, simply use the lowest ϕ possible (between zero and one) such that:

$$\Delta \mathbf{W}_r \cdot \Delta \mathbf{W}_{rg} > 0 \tag{4.9}$$

As long as the dot product is positive, the angle between the vectors will be acute, and the weight change will result in a decrease in E. A ϕ that creates a stable system, in which E is monotonically decreasing, can be found by requiring that the two vectors be orthogonal, then adding any small, positive constant ε to ϕ to convert the right angle into an acute angle:

$$\Delta \mathbf{W}_r \cdot \Delta \mathbf{W}_{rg} = 0$$

$$(\phi \Delta \mathbf{W}_d + (1-\phi)\Delta \mathbf{W}_{rg}) \cdot \Delta \mathbf{W}_{rg} = 0$$

$$\phi = \frac{-\Delta \mathbf{W}_{rg} \cdot \Delta \mathbf{W}_{rg}}{(\Delta \mathbf{W}_d - \Delta \mathbf{W}_{rg}) \cdot \Delta \mathbf{W}_{rg}} \tag{4.10}$$

If this equation yields a ϕ outside the range [0,1], then the direct vector does make an acute angle with the residual gradient vector, so a ϕ of 0 should be used for maximum learning speed. If the denominator of ϕ is zero, this either means that E is at a local minimum, or else it means that the direct algorithm and the residual gradient algorithm yield weight-change vectors pointing in the same direction. In either case, a ϕ of 0 is acceptable. If the equation yields a ϕ between zero and one, then this is the ϕ that causes the mean squared Bellman residual to be constant. Theoretically, any ϕ above this value will ensure convergence. Therefore, a practical implementation of a residual algorithm should first calculate the numerator and denominator separately, then check whether the denominator is zero. If the denominator is zero, then ϕ=0. If it is not, then the algorithm should evaluate equation (4.10), add a small constant ε, then check whether the resulting ϕ lies in the range [0,1]. A ϕ outside this range should be clipped to lie on the boundary of this range.

The above defines residual algorithms in general. For the specific example used in equations (4.6) and (4.7), the corresponding residual algorithm would be:

$$\begin{aligned}\Delta\mathbf{W}_r &= (1-\phi)\Delta\mathbf{W}_d + \phi\Delta\mathbf{W}_{rg} \\ &= -(1-\phi)\alpha\sum_x \left[R+\gamma V(x')-V(x)\right]\left[-\nabla_{\mathbf{W}}V(x)\right] \\ &\quad -\phi\alpha\sum_x \left[R+\gamma V(x')-V(x)\right]\left[\nabla_{\mathbf{W}}\gamma V(x')-\nabla_{\mathbf{W}}V(x)\right] \\ &= -\alpha\sum_x \left[R+\gamma V(x')-V(x)\right]\left[\phi\nabla_{\mathbf{W}}\gamma V(x')-\nabla_{\mathbf{W}}V(x)\right]\end{aligned} \tag{4.11}$$

To implement this incrementally, rather than epoch-wise, the change in a particular weight w after observing a particular state transition would be:

$$\Delta\mathbf{w} = -\alpha\left[R+\gamma V(x')-V(x)\right]\left[\phi\gamma \tfrac{\partial}{\partial w}V(x') - \tfrac{\partial}{\partial w}V(x)\right] \tag{4.12}$$

It is interesting that the residual algorithm turns out to be identical to the residual gradient algorithm in this case, except that one term is multiplied by ϕ.

To find the marginally-stable ϕ using equation (4.10), it is necessary to have an estimate of the epoch-wise weight-change vectors. These can be approximated by maintaining two scalar values, w_d and w_{rg}, associated with each weight w in the function-approximation system. These will be *traces*, averages of recent values, used to approximate $\Delta\mathbf{W}_d$ and $\Delta\mathbf{W}_{rg}$, respectively. The traces are updated according to:

$$\begin{aligned} w_d \longleftarrow (1-\mu)w_d - \mu&\left[R+\gamma V(x')-V(x)\right] \\ &\cdot\left[-\nabla_{\mathbf{W}}V(x)\right]\end{aligned} \tag{4.13}$$

$$w_{rg} \longleftarrow (1-\mu)w_{rg} - \mu\left[R + \gamma V(x') - V(x)\right] \cdot \left[\nabla_{\mathbf{w}} \gamma V(x') - \nabla_{\mathbf{w}} V(x)\right] \tag{4.14}$$

where μ is a small, positive constant that governs how fast the system forgets. A value for ϕ can be found using equation (4.15):

$$\phi = \frac{\sum_{w} w_d \; w_{rg}}{\sum_{w} (w_d - w_{rg}) w_{rg}} + \mu \tag{4.15}$$

If an adaptive ϕ is used, then there is no longer a guarantee of convergence, since the traces will not give perfectly-accurate gradients. Convergence is guaranteed for sufficiently-small ϕ, so a system with an adaptive ϕ might clip it to lie below some user-selected boundary. Or it might try to detect divergence, and decrease ϕ whenever that happens. Adaptive ϕ is just a heuristic.

4.5 Stochastic Markov Chains

The residual algorithm for incremental value iteration in equations (4.12) and (4.15) was derived assuming a deterministic Markov chain.

The derivation above was for a deterministic system. This algorithm does not require that the model of the MDP be known, and it has guaranteed convergence to a local minimum of the mean squared Bellman residual. That is because it would be doing gradient descent on the expected value of a square, rather than a square of an expected value. If the MDP were nondeterministic, then the algorithm would still be guaranteed to converge, but it might not converge to a local minimum of the mean squared Bellman residual. This might still be a useful algorithm, however, because the weights will still converge, and the error in the resulting policy may be small if the MDP is only slightly nondeterministic (deterministic with only a small amount of added randomness).

For a nondeterministic MDP, convergence to a local minimum of the Bellman residual is only guaranteed by using equation (4.16), which also reduces to (4.12) in the case of a deterministic MDP:

$$\Delta w = -\alpha\left[R + \gamma V(x_1') - V(x)\right] \left[\phi\gamma \tfrac{\partial}{\partial w} V(x_2') - \tfrac{\partial}{\partial w} V(x)\right] \tag{4.16}$$

Given a state x, it is necessary to generate two successor states, x_1' and x_2', each drawn independently from the distribution defined by the MDP. This is necessary because an unbiased estimator of the product of two random variables can be obtained by multiplying two independently-generated unbiased estimators. These two independent successor states are easily generated if a model of the MDP is known or is learned. It is also possible to do this without a model, by storing a number of state-successor pairs that are observed, and learning in a given state only after it has been visited twice. This might be particularly useful in a situation where the learning system controls the MDP during learning. If the learning system can intentionally perform actions to return to a given state, then this might be an effective learning method. In any case, it is never necessary to learn the type of detailed, mathematical model of the MDP that would be required by backpropagation through time, and it is never necessary to perform the types of integrals over successor states required by value iteration. It appears that residual algorithms often do not require models of any sort, and on occasion will require only a partial model, which is perhaps the best that can be done when working with completely-general function-approximation systems.

4.6 Extending from Markov Chains to MDPs

Residual algorithms can also be derived for reinforcement learning on MDPs that provide a choice of several actions in each state. The derivation process is the same. Start with a reinforcement learning algorithm that has been designed for use with a lookup table, such as *Q*-learning. Find the equation that is the counterpart of the Bellman equation. This should be an equation whose unique solution is the optimal function that is to be learned. For example, the counterpart of the Bellman equation for *Q*-learning is

$$Q(x,u) = \left\langle R + \gamma \max_{u'} Q(x',u') \right\rangle \tag{4.17}$$

For a given MDP with a finite number of states and actions, there is a unique solution to equation (4.17), which is the optimal Q-function. The equation should be arranged such that the function to be learned appears on the left side, and everything else appears on the right side. The direct algorithm is just backpropagation, where the left side is the output of the network, and the right side is used as the "desired output" for learning. Given the counterpart of the Bellman equation, the mean squared Bellman residual is the average squared difference between the two sides of the equation. The residual gradient algorithm is simply gradient descent on *E*, and the residual algorithm is a weighted sum of the direct and residual gradient algorithms, as defined in equation (4.8).

4.7 Residual Algorithms

Most reinforcement learning algorithms that have been suggested for prediction or control have associated equations that are the counterparts of the Bellman equation. The optimal functions that the learning system should learn are also unique solutions to the Bellman equation counterparts. Given the Bellman equation counterpart for a reinforcement learning algorithm, it is straightforward to derive the associated direct, residual gradient, and residual algorithms. As before, ϕ can be chosen, or it can be adaptive, being calculated in the same way. As can be seen from Table 4.1, all of the residual algorithms can be implemented incrementally except for residual value iteration. Value iteration requires that an expected value be calculated for each possible action, then the maximum to be found. For a system with a continuum of states and actions, a step of value iteration with continuous states would require finding the maximum of an uncountable set of integrals. This is clearly impractical, and appears to have been one of the motivations behind the development of *Q*-learning. Table 4.1 also shows that for a deterministic MDP, all of the algorithms can be implemented without a model, except for residual value iteration. This may simplify the design of a learning system, since there is no need to learn a model of the MDP. Even if the MDP is nondeterministic, the residual algorithms can still be used without a model, by observing x'_1, then using $x'_2=x'_1$. That approximation still ensures convergence, but it may force convergence to an incorrect policy, even if the function-approximation system is initialized to the correct answer, and the initial mean squared Bellman residual is zero. If the nondeterminism is merely a small amount of noise in a control system, then this approximation may be useful in practice. For more accurate results, it is necessary that x'_1 and x'_2 be generated independently. This can be done if a model of the MDP is known or learned, or if the learning system stores tuples (x,u,x'), and then changes the weights only when two tuples are observed with the same x and u. Of course, even when a model of the MDP must be learned, only two successor states need to be generated; there is no need to calculate large sums or integrals as in value iteration.

Table 4.1. Four reinforcement learning algorithms, the counterpart of the Bellman equation for each, and each of the corresponding residual algorithms. The fourth, Advantage learning, is discussed in chapter 5.

Reinforcement Learning Algorithm	Counterpart of Bellman Equation (top) Weight Change for Residual Algorithm (bottom)
TD(0)	$V(x)=\left\langle R+\gamma V(x')\right\rangle$ $\Delta w_r=-\alpha\big(R+\gamma V(x'_1)-V(x)\big)\big(\phi\gamma\frac{\partial}{\partial w}V(x'_2)-\frac{\partial}{\partial w}V(x)\big)$
Value Iteration	$V(x)=\max\limits_u\left\langle R+\gamma V(x')\right\rangle$ $\Delta w_r=-\alpha\big(\max\limits_u\left\langle R+\gamma V(x')\right\rangle-V(x)\big)\big(\phi\frac{\partial}{\partial w}\max\limits_u\left\langle R+\gamma V(x')\right\rangle-\frac{\partial}{\partial w}V(x)\big)$
Q-learning	$Q(x,u)=\left\langle R+\gamma\max\limits_{u'}Q(x',u')\right\rangle$ $\Delta w_r=-\alpha\big(R+\gamma\max\limits_{u'}Q(x'_1,u')-Q(x,u)\big)\big(\phi\gamma\frac{\partial}{\partial w}\max\limits_{u'}Q(x'_2,u')-\frac{\partial}{\partial w}Q(x,u)\big)$
Advantage learning	$A(x,u)=\left\langle R+\gamma^{\Delta t}\max\limits_{u'}A(x',u')\right\rangle\frac{1}{\Delta t}+(1-\frac{1}{\Delta t})\max\limits_{u'}A(x,u')$ $\Delta w_r=-\alpha\Big(\big(R+\gamma^{\Delta t}\max\limits_{u'}A(x'_1,u')\big)\frac{1}{\Delta t}+(1-\frac{1}{\Delta t})\max\limits_{u'}A(x,u')-A(x,u)\Big)$ $\cdot\Big(\phi\gamma^{\Delta t}\frac{\partial}{\partial w}\max\limits_{u'}A(x'_2,u')\frac{1}{\Delta t}+\phi(1-\frac{1}{\Delta t})\frac{\partial}{\partial w}\max\limits_{u'}A(x,u')-\frac{\partial}{\partial w}A(x,u)\Big)$

4.8 Simulation Results

Figure 4.8 shows simulation results. The solid line shows the learning time for the star problem in figure 4.2, and the dotted line shows the learning time for the hall problem in figure 4.5. In the simulation, the direct method was unable to solve the star problem, and the learning time appears to approach infinity as ϕ approaches approximately 0.1034. The optimal constant ϕ appears to lie between 0.2 and 0.3. The adaptive ϕ was able to solve the problem in time close to the optimal time, while the final value of ϕ at the end was approximately the same as the optimal constant ϕ. For the hall problem from figure 4.5, the optimal algorithm is the direct method, ϕ =0. In this case, the adaptive ϕ was able to quickly reach ϕ=0, and therefore solved the problem in close to optimal time. In each case, the learning rate was optimized to two significant digits, through exhaustive search. Each data point was calculated by averaging over 100 trials, each with different initial random weights. For the adaptive methods, the parameters μ=0.001 and ε=0.1 were used, but no attempt was made to optimize them. When adapting, ϕ initially started at 1.0, the safe value corresponding to the pure residual gradient method.

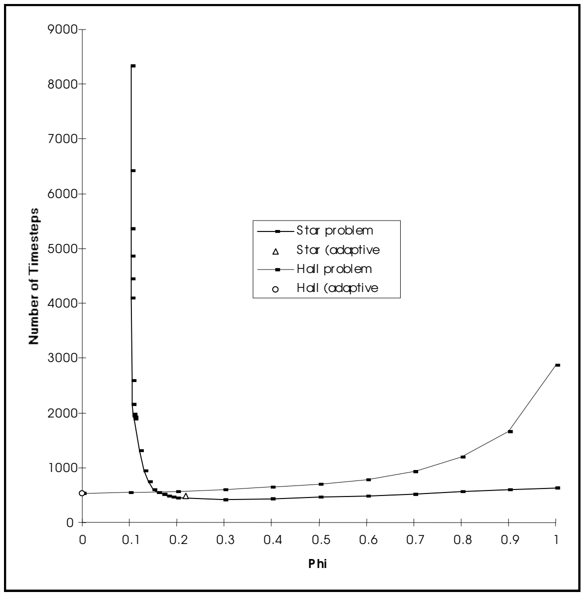

Figure 4.8. Simulation results for two MDPs

The lines in figure 4.8 clearly show that the direct method can be much faster than pure residual gradient algorithms in some cases, yet can be infinitely slower in others. The square and triangle, representing the residual gradient algorithm with adaptive ϕ, demonstrate that the algorithm is able to automatically adjust to the problem at hand and still achieve near-optimal results, at least for these two problems.

Further comparisons of direct and residual algorithms on high-dimensional, nonlinear, problems are given in chapter 5, where the Advantage learning algorithm is proposed. Advantage learning is one example of a residual algorithm

4.9 Summary

Residual algorithms can do reinforcement learning with function approximation systems, with guaranteed convergence, and can learn faster in some cases than both the direct method and pure gradient descent (*residual gradient* algorithms). Local minima have not been a problem for the problems shown here. The shortcomings of both direct and residual gradient algorithms have been shown. It has also been shown, both analytically and in simulation, that direct and residual gradient algorithms are special cases of residual algorithms, and that residual algorithms can be found that combine the beneficial properties of both. This allows reinforcement learning to be combined with general function-approximation systems, with fast learning, while retaining guaranteed convergence.

5 Advantage learning: Learning with Small Time Steps

Q-learning is sometimes preferable to value iteration, such as in some problems that are highly stochastic and poorly modele. Often, these problems deal with continuous time, such as in some robotics and control problems, and differential games. Unfortunately, *Q*-learning with typical function approximators is unable to learn anything useful in these problems. This chapter introduces a new algorithm, Advantage learning, which is exponentially faster than *Q*-learning for continuos-time problems. It is interesting that this algorithm is one example of a residual algorithm, as defined in chapter 4. In fact, the direct form of the algorithm wouldn't even have the convergence results that exist for *Q*-learning on lookup tables. The contribution of this chapter therefore illustrates the usefulness of the gradient-descent concept, as shown in chapter 4.

The empirical results are also interesting, as they involve a 6-dimensional, real-valued state space, highly nonlinear, nonholonomic dynamics, continuous time, and optimal game playing rather than just control. The results show a dramatic advantage of Advantage learning over *Q*-learning (the latter couldn't learn at all), and residual algorithms over direct (the latter couldn't learn at all).

5.1 Introduction

In work done before the development of the residual algorithms, an algorithm called *advantage updating* (Harmon, Baird, and Klopf, 1995) was proposed that seemed preferable to *Q*-learning for continuous-time systems. It was shown in simulation that it could learn the optimal policy for a linear-quadratic differential game using a quadratic function approximation system. Unfortunately, it required two function approximators rather than one, and there was no convergence proof for it, even for lookup tables. In fact, under some update sequences (though not those suggested in the paper), it could be shown to oscillate forever between the best and worst possible policies. This result came from essentially forcing it to act like the actor-critic system in figure 2.2. This was an unfortunate result, since in simulation it learned the optimal policy exponentially faster than *Q*-learning as the time-step size was decreased. It was never clear how the algorithm could be extended to solve its theoretical problems, nor was it clear how it could be analyzed. This particular problem was actually the original motivation behind the development of residual algorithms, described in chapter 4.

In this chapter, a new algorithm is derived: *advantage learning*, which retains the good properties of advantage updating but requires only one function to be learned rather than two, and which has guaranteed convergence to a local optimum. It is a residual algorithm, so both the derivation and the analysis are much simpler than for the original algorithm. This illustrates the power of the general gradient-descent concept for developing and analyzing new reinforcement-learning algorithms.

This chapter derives the advantage learning algorithm and gives empirical results demonstrating it solving a non-linear game using a general neural network. The game is a Markov decision process (MDP) with continuous states and non-linear dynamics. The game consists of two players, a missile and a plane; the missile pursues the plane and the plane evades the missile. On each time step, each player chooses one of two possible actions; turn left or turn right 90 degrees. Reinforcement is given only when the missile either hits or misses the plane, which makes the problem difficult. The advantage function is stored in a single-hidden-layer sigmoidal network. The reinforcement learning algorithm for optimal control is modified for differential games in order to find the minimax point, rather than the maximum. This is the first time that a reinforcement learning algorithm with guaranteed convergence for general function approximation systems has been demonstrated to work with a general neural network.

5.2 Background

5.2.1 Advantage updating

The advantage updating algorithm (Baird, 1993) is a reinforcement learning algorithm in which two types of information are stored. For each state x, the value $V(x)$ is stored, representing an estimate of the total discounted return expected when starting in state x and performing optimal actions. For each state x and action u, the *advantage*, $A(x,u)$, is stored, representing an estimate of the degree to which the expected total discounted reinforcement is increased by performing action u rather than the action currently considered best. It might be called the negative of regret for that action. The optimal value function $V^*(x)$ represents the true value of each state. The optimal advantage function $A^*(x,u)$ will be zero if u is the optimal action (because u confers no advantage relative to itself) and $A^*(x,u)$ will be negative for any suboptimal u (because a suboptimal action has a negative advantage relative to the best action).

Advantage updating has been shown to learn faster than Q-learning, especially for continuous-time problems (Baird, 1993, Harmon, Baird, & Klopf, 1995). It is not a residual algorithm, though, so there is no proof of convergence, even for lookup tables, and there is no obvious way to reduce its requirements from two function approximators to one.

5.2.2 Advantage learning

Advantage learning improves on advantage updating by requiring only a single function to be stored, the advantage function $A(x,u)$, which saves memory and computation. It is a residual algorithm, and so is guaranteed to converge to a local minimum of mean squared Bellman residual. Furthermore, advantage updating requires two types of updates (learning and normalization updates), while advantage learning requires only a single type of update (the learning update). For each state-action pair (x,u), the advantage $A(x,u)$ is stored, representing the utility (advantage) of performing action u rather than the action currently considered best. The optimal advantage function *A*(x,u)* represents the true advantage of each state-action pair. The value of a state is defined as:

$$V^*(x) = \max_u A^*(x,u) \tag{5.1}$$

The advantage *A*(x,u)* for state *x* and action *u* is defined to be:

$$A^*(x,u) = V^*(x) + \frac{\left\langle R + \gamma^{\Delta t} V^*(x') \right\rangle - V^*(x)}{\Delta t K} \tag{5.2}$$

where $\gamma^{\Delta t}$ is the discount factor per time step, *K* is a time unit scaling factor, and < > represents the expected value over all possible results of performing action *u* in state *x* to receive immediate reinforcement *R* and to transition to a new state *x'*. Under this definition, an advantage can be thought of as the sum of the value of the state plus the expected rate at which performing *u* increases the total discounted reinforcement. For optimal actions the second term is zero; for suboptimal actions the second term is negative. Note that in advantage learning, the advantages are slightly different than in advantage updating. In the latter, the values were stored in a separate function approximator. In the former, the value is part of the definition, as seen in equation (5.2).

The Advantage function can also be written in terms of the optimal Q function, as in equation (5.3).

$$A^*(x,u) = V^*(x) - \frac{V^*(x) - Q^*(x,u)}{\Delta t K} \tag{5.3}$$

which suggests a simple graphical representation of how Advantages compare to Q values, shown in figure 5.2.

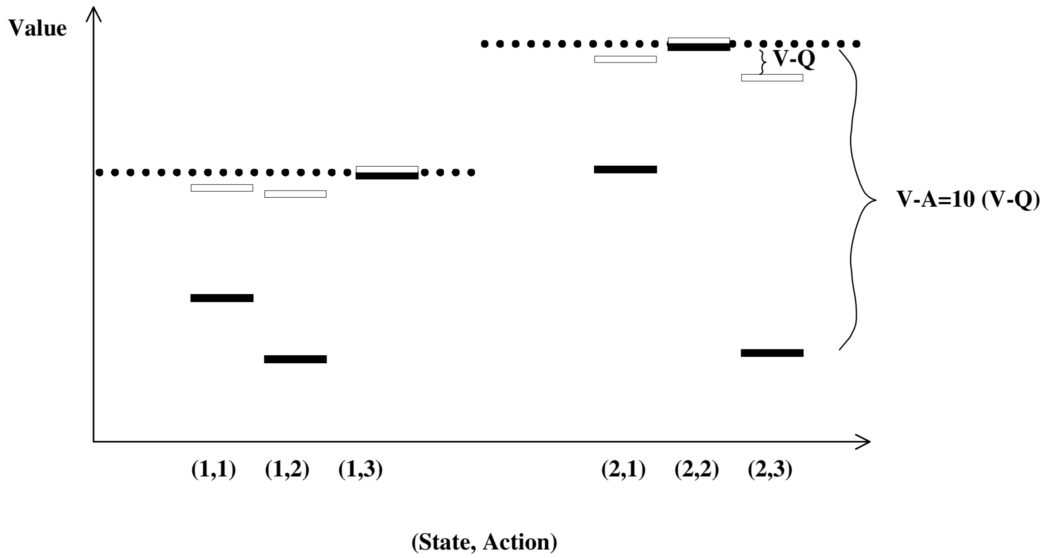

Figure 5.1. Comparison of Advantages (black) to Q values (white) in the case that $1/(k\Delta t)$=10. The dotted line in each state represents the value of the state, which equals both the maximum Q value and the maximum Advantage. Each A is 10 times as far from V as the corresponding Q.

In figure 5.1, The Q values (white) are close together in each state, but differ greatly from state to state. During Q learning with a function approximator, small errors in the Q values will have large effects on the policy. The Advantages (black) are well distributed, and small errors in them will not greatly affect the policy. As Δt shrinks, the Q values all approach their respective dotted lines, while the Advantages do not move. All of this is similar to what happened in Advantage updating, but in Advantage learning it is simpler, since there is no need to store a separate value function. And the latter is guaranteed to converge. Not surprisingly, learning can be much faster than Q learning, as can be seen by comparing the algorithms on a linear quadratic regulator (LQR) problem..

The LQR problem is as follows. At a given time t, the state of the system being controlled is the real value x_t. The controller chooses a control action u_t which is also a real value. The dynamics of the system are:

$$\dot{x}_t = u_t$$

The rate of reinforcement to the learning system, $r(x_t,u_t)$, is

$$r(x_t,u_t) = -x_t^2 - u_t^2$$

Given some positive discount factor γ<1, the goal is to maximize the total discounted reinforcement:

$$\int_0^{\infty} \gamma^t r(x_t, u_t)\,dt$$

A discrete-time controller can change its output every Δt seconds, and its output is constant between changes. The discounted reinforcement received during a single time step is:

$$R_{\Delta t}(x_t, u_t) = \int_t^{t+\Delta t} \gamma^{\tau - t} r(x_\tau, u_\tau)\,d\tau = \int_t^{t+\Delta t} \gamma^{\tau - t}\left(-(x_\tau + \tau u_\tau)^2 - u_\tau^2\right)d\tau$$

and the total reinforcement to be maximized is:

$$\sum_{i=0}^{\infty} \left(\gamma^{\Delta t}\right)^i R_{\Delta t}(x_{i\Delta t}, u_{i\Delta t})$$

The results of comparison experiments on this LQR problem are in figure 5.1:

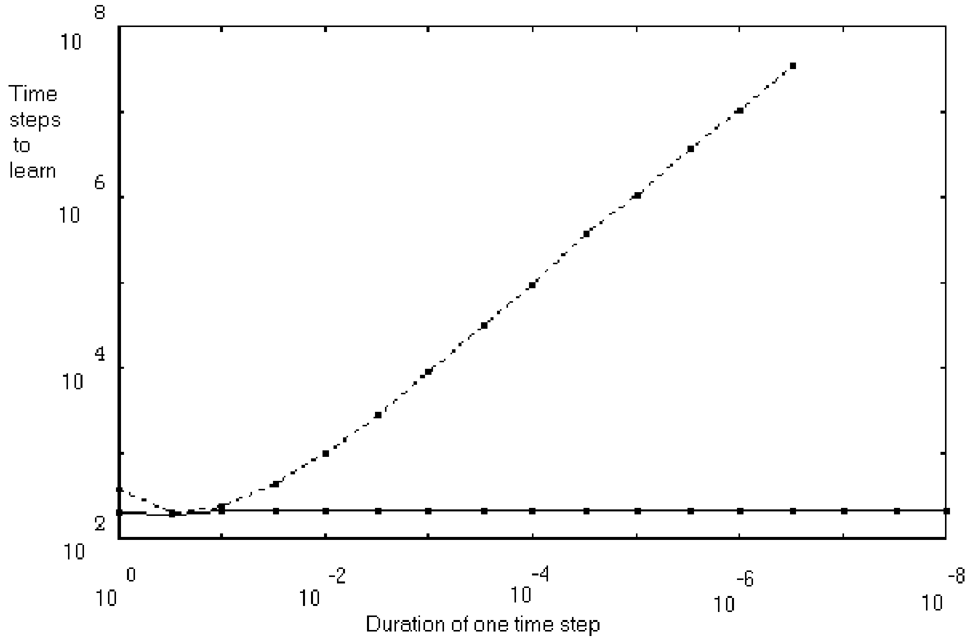

Figure 5.2. Advantages allow learning whose speed is independent of the step size, while Q learning learns much slower for small step sizes.

When Δt=1, it reduces to Q learning, and so takes the same amount of time to learn. As Δt goes to zero, the target Advantage function A^* does not change, while the target Q function Q^* becomes almost flat in the action direction. That makes Q very susceptible to noise, and causes it to take much longer to learn.

5.3 Reinforcement Learning with Continuous States

5.3.1 Direct Algorithms

For predicting the outcome of a Markov chain (a degenerate MDP for which there is only one possible action in each state), an obvious algorithm is an incremental form of value iteration, which is defined as:

$$V(x) \leftarrow (1-\alpha)V(x) + \alpha[R + \gamma V(x')] \tag{5.4}$$

If V(x) is represented by a function-approximation system other than a look-up table, update (5.4) can be implemented directly by combining it with the backpropagation algorithm (Rumelhart, Hinton, & Williams, 86). For an input x, the output of the function-approximation system would be V(x), the "desired output" used for training would be $R+\gamma V(x')$, and all of the weights would be adjusted through gradient descent to make the actual output closer to the desired output. Equation (5.4) is exactly the TD(0) algorithm, and could also be called the *direct* implementation of incremental value iteration, Q-learning, and advantage learning.

5.3.2 Residual Gradient Algorithms

Reinforcement learning algorithms can be guaranteed to converge for lookup tables, yet be unstable for function-approximation systems that have even a small amount of generalization when using the direct implementation (Boyan, 95). To find an algorithm that is more stable than the direct algorithm, it is useful to specify the exact goal for the learning system. For the problem of prediction on a deterministic Markov chain, the goal can be stated as finding a value function such that, for any state x and its successor state x', with a transition yielding immediate reinforcement R, the value function will satisfy the Bellman equation:

$$V(x) = \langle R + \gamma V(x') \rangle \tag{5.5}$$

For a given value function V, and a given state x, the *Bellman residual* is defined to be the difference between the two sides of the Bellman equation. The *mean squared Bellman residual* for an MDP with n states is therefore defined to be:

$$E = \frac{1}{n} \sum_x \left[\langle R + \gamma V(x') \rangle - V(x) \right]^2 \tag{5.6}$$

Residual gradient algorithms change the weights in the function-approximation system by performing gradient descent on the mean squared Bellman residual, E. This is called the *residual gradient* algorithm. The residual gradient algorithm and a faster version called the *residual* algorithm are described in chapter 4.

The counterpart of the Bellman equation for advantage learning is:

$$A^*(x,u) = \rangle R + \gamma^{\Delta t} A^*_{\max_{u'}}(x',u') \langle \frac{1}{\Delta t K} + \left(1 - \frac{1}{\Delta t K}\right) A^*_{\max_{u}}(x,u') \tag{5.7}$$

If $A(x,u)$ is an approximation of $A^*(x,u)$, then the mean squared Bellman residual, E, is:

$$E = \left\rangle \left(\rangle R + \gamma^{\Delta t} A_{\max_{u'}}(x',u') \langle \frac{1}{\Delta t K} + \left(1 - \frac{1}{\Delta t K}\right) A_{\max_{u'}}(x,u') - A(x,u') \right)^2 \right\langle \tag{5.8}$$

where the inner <> is the expected value over all possible results of performing a given action u in a given state x, and the outer <> is the expected value over all possible states and actions.

5.4 Differential Games

Differential games (Isaacs, 1965) are played in continuous time, or use sufficiently small time steps to approximate continuous time. Both players evaluate the given state and simultaneously execute an action, with no knowledge of the other player's selected action.

The *value* of a game is the long-term, discounted reinforcement if both opponents play the game optimally in every state. Consider a game in which player A tries to minimize the total discounted reinforcement, while the opponent, player B, tries to maximize the total discounted reinforcement. Given the advantage $A(x,u_A,u_B)$ for each possible action in state x, it is useful to define the minimax and maximin values for state x as:

$$\text{minimax}(x) = \min_{u_A} \max_{u_B} A\left(x, u_A, u_B\right) \tag{5.9}$$

$$\text{maximin}(x) = \max_{u_B} \min_{u_A} A\left(x, u_A, u_B\right) \tag{5.10}$$

If the minimax equals the maximin, then the minimax is called a *saddlepoint* and the optimal policy for both players is to perform the actions associated with the saddlepoint. If a saddlepoint does not exist, then the optimal policy is stochastic if an optimal policy exists at all. If a saddlepoint does not exist, and a learning system treats the minimax as if it were a saddlepoint, then the system will behave as if player A must choose an action on each time step, and then player B chooses an action based upon the action chosen by A. For the algorithms described below, a saddlepoint is assumed to exist. If a saddlepoint does not exist, this assumption gives a slight advantage to player B.

5.5 Simulation of the Game

5.5.1 Advantage learning

During training, a state is chosen from a uniform random distribution on each learning cycle. The vector of weights in the function approximation system, **W,** is updated according to equation (5.11) on each time step.

$$\begin{aligned}\Delta W = -\alpha&\left(\left(R+\gamma^{\Delta t} A_{\min\max}(x',u)\right)\frac{1}{\Delta t K} + \left(1-\frac{1}{\Delta t K}\right)A_{\min\max}(x,u) - A(x,u)\right)\\ &\bullet\left(\phi\gamma^{\Delta t}\frac{\partial}{\partial \mathbf{W}}A_{\min\max}(x',u)\frac{1}{\Delta t K} + \phi\left(1-\frac{1}{\Delta t K}\right)\frac{\partial}{\partial \mathbf{W}}A_{\min\max}(x,u) - \frac{\partial}{\partial \mathbf{W}}A(x,u)\right)\end{aligned} \tag{5.11}$$

The parameter ϕ is a constant that controls a trade-off between pure gradient descent (when ϕ equals 0) and a fast direct algorithm (when ϕ equals 1). ϕ can change adaptively by calculating two values, w_d and w_{rg}. These are *traces*, averages of recent values, updated according to:

$$w_d \leftarrow (1-\mu)w_d - \mu\left[\left(R+\gamma^{\Delta t}A_{\min\max}(x',u)\right)/\Delta tK + (1-1/\Delta tK)A_{\min\max}(x,u)\right] \bullet \left[-\frac{\partial}{\partial w}A_{\min\max}(x,u)\right] \tag{5.12}$$

$$w_{rg} \leftarrow (1-\mu)w_{rg} - \mu\begin{bmatrix}\left(R+\gamma^{\Delta t}A_{\min\max}(x',u)\right)/\Delta tK + \\ (1-1/\Delta tK)A_{\min\max}(x,u) - A(x,u)\end{bmatrix} \bullet \begin{bmatrix}\left(\gamma^{\Delta t}\frac{\partial}{\partial w}A_{\min\max}(x',u)\right)/(\Delta tK) + \\ (1-1/\Delta tK)\frac{\partial}{\partial w}A_{\min\max}(x,u) - \frac{\partial}{\partial w}A(x,u)\end{bmatrix} \tag{5.13}$$

where μ was a small, positive constant that governed how fast the system forgets. On each time step a stable ϕ is calculated by using equation (5.14). This ensures convergence while maintaining fast learning:

$$\phi = \frac{\sum_w w_d w_{rg}}{\sum_w (w_d - w_{rg})w_{rg}} + \mu \tag{5.14}$$

5.5.2 Game Definition

The problem is a differential game with a missile pursuing a plane, similar to other pursuit games (Rajan, Prasad, and Rao, 1980, Millington 1991). The action performed by the missile is a function of the state, which is the position and velocity of both players. The action performed by the plane is a function of the state and the action of the missile.

The game is a Markov game with continuous states and non-linear dynamics. The state **x** is a vector ($\mathbf{x}_m$,$\mathbf{x}_\mathrm{p}$) composed of the state of the missile and the state of the plane, each of which are composed of the position and velocity of the player in two-dimensional space. The action **u** is a vector (u_m,u_p) composed of the action performed by the missile and the action performed by the plane, each of which is a scalar value; 0.5 indicates a 90 degree turn to the left, and -0.5 indicates a 90 degree turn to the right. The next state $\mathbf{x}_{t+1}$ is a non-linear function of the current state $\mathbf{x}_t$ and action $\mathbf{u}_t$. The speed of each player is fixed, with the speed and turn radius of the missile twice that of the plane. On each time step the heading of each player is updated according to the action chosen, the velocity in both the x and y dimensions is computed for each player, and the positions of the players are updated.

The reinforcement function *R* is a function of the distance between the players. A reinforcement of 1 is given when the Euclidean distance between the players grows larger than 2 units (plane escapes). A reinforcement of -1 is given when the distance grows smaller than 0.25 units (missile hits plane). No reinforcement is given when the distance is in the range [0.25,2]. The missile seeks to minimize reinforcement, while the plane seeks to maximize reinforcement.

The advantage function is approximated by a single-hidden-layer neural network with 50 hidden nodes. The hidden-layer nodes each have a sigmoidal activation function whose output lies in the range [-1,1]. The output of the network is a linear combination of the outputs of the hidden-layer nodes with their associated weights. To speed learning a separate adaptive learning rate was used for each weight in the network. There are 6 inputs to the network. The first 4 inputs describe the state and are normalized to the range [-1,1]. They consist of the difference in positions and velocities of the players in both the x and y dimensions. The remaining inputs describe the action to be taken by each player; 0.5 and -0.5 indicate left and right turns respectively.

5.6 Results

Experiments were formulated to accomplish two objectives. The first objective was to determine to what degree advantage learning could learn the optimal policy for the missile/aircraft system. The second objective was to compare the performances of advantage learning when implemented in the residual gradient form, in the direct form, and using weighted averages of the two by using values of ϕ in the range [0,1].

In Experiment 1, the residual form of advantage learning learned the correct policy after 800,000 training cycles. The missile learned to pursue the plane, and the plane learned to evade the missile. Interesting behavior was exhibited by both players under certain initial conditions. First, the plane learned that in some cases it is able to indefinitely evade the missile by continuously flying in circles within the missile's turn radius. Second, the missile learned to anticipate the position of the plane. Rather than heading directly toward the plane, the missile learned to lead the plane under appropriate circumstances.

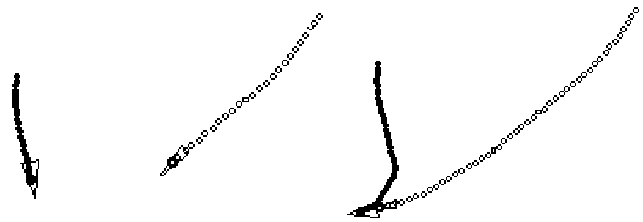

Figure 5.3. The first snapshot (pictures taken of the actual simulator) demonstrates the missile leading the plane and, in the second snapshot, ultimately hitting the plane.

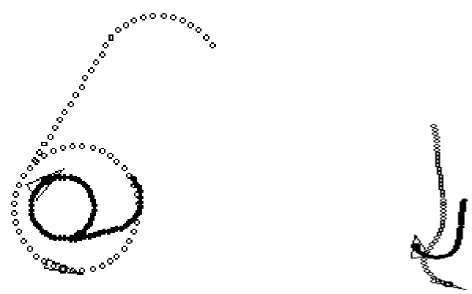

Figure 5.4. The first snapshot demonstrates the ability of the plane to survive indefinitely by flying in continuous circles within the missile's turn radius. The second snapshot demonstrates the learned behavior of the plane to turn toward the missile to increase the distance between the two in the long term, a tactic used by pilots.

In Experiment 2, different values of ϕ were used for the weighting factor in residual advantage learning. Six different experiments were run, each using identical parameters with the exception of the weighting factor ϕ. Figure 5.5 presents the results of these experiments. The dashed line is the error level after using an adaptive ϕ. A ϕ of 1 yields advantage learning in the residual gradient form, while a ϕ of 0 yields advantage learning implemented in the direct form.

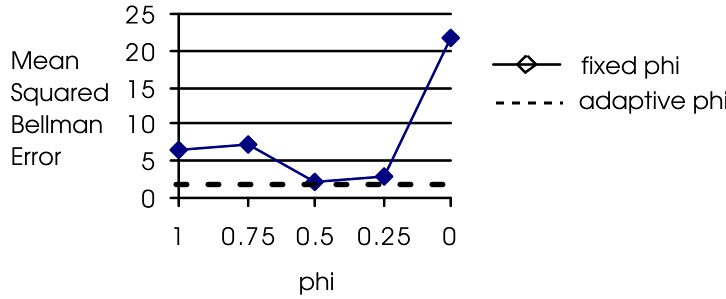

Figure 5.5: ϕ comparison. Final Bellman error after using various values of the fixed ϕ (solid), or using the adaptive ϕ (dotted).

5.7 Summary

This chapter shows the power of the gradient-descent concept by deriving a new, residual algorithm. Advantage updating was a useful algorithm, though it had no convergence proof, and was inelegant. Residual algorithms allowed the development of Advantage learning, which was the same as Advantage updating in practice, but had better theoretical properties, and also used less computational and memory resources. It was compared on a test problem that is highly non-linear, with continuous states. In general, non-linear problems of this type are difficult to solve with classical game theory and control theory, and therefore appear to be good applications for reinforcement learning. It was shown that the residual algorithm with adaptive ϕ was able to perform as well as with the optimal ϕ. Furthermore, the policy learned by the system yielded behavior resembling the strategies used by pilots. Neither *Q*-learning nor a direct form of Advantage learning was able to learn anything at all, which suggests both the utility of Advantage learning for continuous time, and the utility of residual algorithms in general.

6 VAPS: Value and Policy Search, and Guaranteed Convergence for Greedy Exploration

This chapter proposes VAPS, a generalization of residual algorithms that allows the exploration policy to change during learning. In addition, it allows a learning system to forget about values altogether, and just search in policy space directly. In the field of reinforcement learning, algorithms that use values tend to be very separate from those that do policy search, so it is surprising that a single family of algorithms could do both. However, VAPS is a gradient-descent algorithm, and any two gradient-descent algorithms can always be combined by summing their error functions. Therefore, it can handle both Values And Policy Search (VAPS) simultaneously, with just a single function approximator, not a separate ones for values and policies. This result is only possible because the algorithms are derived from first principles using a gradient-descent technique. Simulation results suggest that it is useful to combine these two approaches that have traditionally resided in entirely different camps.

6.1 Convergence Results

Many reinforcement-learning algorithms are known that use a parameterized function approximator to represent a value function, and adjust the weights incrementally during learning. Examples include Q-learning, SARSA, and advantage learning (chapter 5). There are simple MDPs where the original form of these algorithms fails to converge, as demonstrated in chapter 3, and summarized in Table 6.1. For the cases with √, the algorithms are guaranteed to converge under reasonable assumptions such as decaying learning rates. For the cases with **X**, there are known counterexamples where it will either diverge or oscillate between the best and worst possible policies, which have values that are very different. This can happen even with infinite training time and slowly-decreasing learning rates (Baird, 95, Gordon, 96). If a box on the chart contains an **X**, then it will never be possible to prove that all situations in that box avoid disaster. It may be possible, however, that future research will prove that some subset of the box does have guaranteed convergence or guaranteed avoidance of disastrous oscillations. Perhaps new classes of function approximators, or particular types of MDPs will be shown to have this property. At the moment, though, the chart reflects the results that are known.

Table 6.1 has three columns, corresponding to three types of training example distributions. In a *fixed distribution*, each transition is seen a certain percentage of the time. This might be done by drawing transitions randomly from a database of previously-recorded transitions. It could also be done by training on transitions as they are seen while following a fixed, stochastic exploration policy. It might even be done by sorting all possible transitions into some order, then making a sweep through the list, then resorting and repeating, as in prioritized sweeping.

In *on-policy* training, the fixed distribution corresponds to how often the transitions are seen while following a fixed, stochastic policy. This restriction on the distribution allows convergence to be guaranteed for pure prediction problems (Markov Chains), with linear function approximators. Unfortunately, it doesn't allow convergence proofs for any other case on the chart.

In a mostly-greedy distribution, the transitions are generated by following trajectories. These trajectories are generated by following a stochastic policy, but the policy itself changes over time as a result of learning. This will be the case for almost any real-world reinforcement-learning problem, since it is generally useful to train on the same types of states during learning as those that will be seen when using the learned policy. If learning is done entirely with some fixed policy, then the learned policy is likely to be different, and the two are likely to spend time in different regions of state space. That is why it is usually necessary to allow the policy to change. For a very stochastic problem like backgammon, the policy can simply be *greedy*: during learning, the system can always choose the action which appears optimal according its current value function. For a more deterministic problem, like chess, it would be better to be *mostly-greedy*, occasionally choosing actions that are not greedy with respect to the current values, just to ensure sufficient exploration. Unfortunately, this third column has known counterexamples in almost every case.

The rows of the chart or divided into three sections: Markov chains, MDPs, and POMDPs. Markov chains are pure prediction, with no policy, so there is no entry for usually-greedy distributions on Markov chains. The MDP rows are for problems with the Markov property, where the next state distribution depends only on the current state and action. The POMDP rows are for those problems lacking this property.

Within each type of problem, there are four rows corresponding to different types of function approximators. Lookup tables are simple, and have guaranteed convergence for MDPs and Markov chains. For POMDPs, however, even lookup tables are not guaranteed to converge with existing algorithms (Gordon, 96). For linear function approximators, where the value is a linear function of the weights and a possibly-nonlinear function of the states and actions, there are diverging counterexamples for most cases. If the problem is pure prediction (a Markov chain) and the distribution is on-policy, then convergence is guaranteed (Sutton, 1988). In all other cases, there are counterexamples that diverge. For nonlinear function approximators, even on-policy training can diverge on pure prediction problems (Tsitsiklis & Van Roy, 1997). General, nonlinear function approximators can diverge in every case. Since even linear function approximators can diverge for MDPs and POMDPs, more general approximators such as neural networks can also diverge. There is, however, a class of function approximators that have guaranteed convergence for MDPs, though not POMDPs (Gordon, 1999). These *averagers* are systems such as *K*-nearest neighbors where the output for a given input is an average of the outputs of stored data. It would not include locally-weighted regression, where an extrapolated value can be greater than all of the data points. It is also important to note that the chart only refers to incremental algorithms using value functions that slowly change weights in a function approximator. It does not include algorithms that solve MDPs directly using linear programming (Gordon, 1999), or pure policy-search methods such as backpropagation through time.

It is interesting that the chart gives the same result for MDPs with linear function approximators as for POMDPs with lookup tables. That is because these two cases are actually the same situation. A linear function approximator can be viewed as a system that first performs some possibly-nonlinear function on the state or state-action pair, possibly even changing their dimensionality. Then, it uses a lookup table on the transformed state. If the initial, fixed transformation is thought of as part of the environment rather than part of the learning system, then this problem reduces to lookup tables on POMDPs.

Each **X** in the first two columns can be changed to a √ and made to converge by using a modified form of the algorithm, the *residual* form described in chapter 4. However, this is only possible when learning with a fixed training distribution, and that is rarely practical. For most large problems, it is useful to explore with a policy that is usually-greedy with respect to the current value function, and that changes as the value function changes. In that case (the rightmost column of the chart), the current convergence guarantees are not very good.

One way to guarantee convergence in all three columns is to modify the algorithm so that it is performing stochastic gradient descent on some average error function, where the average is weighted by state-visitation frequencies for the current usually-greedy policy. Then the weighting changes as the policy changes. It might appear that this gradient is difficult to compute. Consider Q-learning exploring with a Boltzman distribution that is usually greedy with respect to the learned Q function. It seems difficult to calculate gradients, since changing a single weight will change many Q values, changing a single Q value will change many action-choice probabilities in that state, and changing a single action-choice probability may affect the frequency with which every state in the MDP is visited. Although this might seem difficult, it is not. Surprisingly, unbiased estimates of the gradients of visitation distributions with respect to the weights can be calculated quickly, and the resulting algorithms can put a √ in every case in Table 6.1.

Table 6.1. Current convergence results for incremental, value-based RL algorithms. Residual algorithms changed every X in the first two columns

		Fixed distribution (on-policy)	Fixed distribution	Usually-greedy distribution
Markov chain	Lookup table	√	√	
	Averager	√	√	
	Linear	√	**X**	
	Nonlinear	**X**	**X**	
MDP	Lookup table	√	√	√
	Averager	√	√	**X**
	Linear	**X**	**X**	**X**
	Nonlinear	**X**	**X**	**X**
POMDP	Lookup table	**X**	**X**	**X**
	Averager	**X**	**X**	**X**
	Linear	**X**	**X**	**X**
	Nonlinear	**X**	**X**	**X**

√=convergence guaranteed
X=counterexample is known that either diverges or oscillates between the best and worst possible policies.

to √. The new VAPS form of the algorithms changes every X to a √.

6.2 Derivation of the VAPS equation

Consider a sequence of transitions observed while following a particular stochastic policy on an MDP. Let $s_t = \{x_0,u_0,R_0,\ x_1,u_1,R_1,\ \ldots\ x_{t-1},u_{t-1},R_{t-1},\ x_t,u_t,R_t\}$ be the sequence of states, actions, and reinforcements up to time t, where performing action u_i in state x_i yields reinforcement R_i and a transition to state x_{i+1}. The stochastic policy may be a function of a vector of weights **w**. Without loss of generality, assume the MDP has a single start state named x_0. If the MDP has terminal states, and x_i is a terminal state, then $x_{i+1}=x_0$. Let $\mathbf{S}_t$ be the set of all possible sequences from time 0 to t. Let $e(s_t)$ be a given error function that calculates an error on each time step, such as the squared Bellman residual at time t, or some other error occurring at time t. Note that the error at time t can potentially be a function of everything that has happened so far on the sequence. If e is a function of the weights, then it must be a smooth function of the weights. Consider a period of time starting at time 0 and ending with probability $P(end \mid s_t)$ after the sequence s_t occurs. The probabilities must be such that the expected squared period length is finite. Let B be the expected total error during that period, where the expectation is weighted according to the state-visitation frequencies generated by the given policy:

$$B = \sum_{T=0}^{\infty} \sum_{s_T \in \mathbf{S}_T} P(\text{period ends at time } T \text{ after trajectory } \mathrm{S_T}) \sum_{s_i} e(s_i)$$

$$= \sum_{t=0}^{\infty} \sum_{s_t \in \mathbf{S}_t} e(s_t) P(s_t) \tag{6.1}$$

where s_T is a sequence from time 0 to time T, $\mathbf{S}_T$ is the set of all possible such sequences, the inner summation is over each s_i which is a subsequence of s_T going from time 0 to time i (for all i from 0 to T), and P is:

$$P(s_t) = P(u_t \mid s_t) P(R_t \mid s_t) \prod_{i=0}^{t-1} P(u_i \mid s_i) P(R_i \mid s_i) P(x_{i+1} \mid s_i) [1 - P(end \mid s_i)] \tag{6.2}$$

Note that on the first line, for a particular s_t, the error $e(s_t)$ will be added in to B once for every sequence that starts with s_t. Each of these terms will be weighted by the probability of a complete trajectory that starts with s_t. The sum of the probabilities of all trajectories that start with s_t is simply the probability of s_t being observed, since the period is assumed to end eventually with probability one. So ref **Error! Bookmark not defined.** equals (6.1). Then (6.2) is the probability of the sequence, of which only the $P(u_i|s_i)$ factor might be a function of **w**. If so, this probability must be a smooth function of the weights and nonzero everywhere. The partial derivative of B with respect to w, a particular element of the weight vector **w**, is:

$$\frac{\partial}{\partial w} B = \sum_{t=0}^{\infty} \sum_{s_t \in \mathbf{S}_t} \left(\left(\frac{\partial}{\partial w} e(s_t) \right) P(s_t) + e(s_t) P(s_t) \sum_{j=1}^{t} \frac{\frac{\partial}{\partial w} [P(u_{j-1} \mid s_{j-1}]}{P(u_{j-1} \mid s_{j-1})} \right)$$

$$= \sum_{t=0}^{\infty} \sum_{s_t \in \mathbf{S}_t} P(s_t) \left[\tfrac{\partial}{\partial w} e(s_t) + e(s_t) \sum_{j=1}^{t} \tfrac{\partial}{\partial w} \ln\left(P(u_{j-1} \mid s_{j-1})\right) \right] \tag{6.3}$$

Summing (6.3) over an entire period gives an unbiased estimate of *B*, the expected total error during a period. An incremental algorithm to perform stochastic gradient descent on *B* is the weight update given on the left side of Table 6.2, where the summation over previous time steps is replaced with a trace T_t for each weight. This algorithm is more general than previous algorithms of this form, in that *e* can be a function of all previous states, actions, and reinforcements, rather than just the current reinforcement. This is what allows VAPS to do both value and policy search.

Every algorithm proposed in this chapter is a special case of the VAPS equation on the left side of Table 6.2. Note that no model is needed for this algorithm. The only probability needed in the algorithm is the policy, not the transition probability from the MDP. This is stochastic gradient descent on *B*, and the update rule is only correct if the observed transitions are sampled from trajectories found by following the current, stochastic policy. Both *e* and *P* should be smooth functions of **w**, and for any given **w** vector, *e* should be bounded. The algorithm is simple, but actually generates a large class of different algorithms depending on the choice of *e* and when the trace is reset to zero. For a single sequence, sampled by following the current policy, the sum of Δw along the sequence will give an unbiased estimate of the true gradient, with finite variance. Therefore, during learning, if weight updates are made at the end of each trial, and if the weights stay within a bounded region, and the learning rate approaches zero, then *B* will converge with probability one. Adding a weight-decay term (a constant times the 2-norm of the weight vector) onto *B* will prevent weight divergence for small initial learning rates. There is no guarantee that a global minimum will be found when using general function approximators, but at least it will converge. This is true for backpropagation as well.

Table 6.2. The general VAPS algorithm (left), and several instantiations of it (right). This single algorithm includes both value-based and policy-search approaches and their combination, and gives guaranteed convergence in every case.

<table>
<tr><td rowspan="5">$\Delta w_t = -\alpha\left[\frac{\partial}{\partial w} e(s_t) + e(s_t) T_t\right]$

$\Delta T_t = \frac{\partial}{\partial w} \ln(P(u_{t-1} \mid s_{t-1}))$</td><td>$e_{SARSA}(s_t) = \frac{1}{2} E^2\left[R_{t-1} + \gamma Q(x_t, u_t) - Q(x_{t-1}, u_{t-1})\right]$</td></tr>
<tr><td>$e_{Q-learning}(s_t) = \frac{1}{2} E^2\left[R_{t-1} + \gamma \max_u Q(x_t, u) - Q(x_{t-1}, u_{t-1})\right]$</td></tr>
<tr><td>$e_{advantage}(s_t) = \frac{1}{2} E^2\left[\begin{array}{r} R_{t-1} + \gamma \max_u A(x_t, u) - \frac{\Delta t}{K} A(x_{t-1}, u_{t-1}) \\ + \left(\frac{\Delta t}{K} - 1\right) \max_u A(x_{t-1}, u) \end{array}\right]$</td></tr>
<tr><td>$e_{value-iteration}(s_t) = \frac{1}{2}\left[\max_{u_{t-1}} E\left[R_{t-1} + \gamma V(x_t)\right] - V(x_{t-1})\right]^2$</td></tr>
<tr><td>$e_{SARSA-policy}(s_t) = (1 - \beta) e_{SARSA}(s_t) + \beta\left(b - \gamma^t R_t\right)$</td></tr>
</table>

6.3 Instantiating the VAPS Algorithm

Many reinforcement-learning algorithms are *value-based*; they try to learn a value function that satisfies the Bellman equation. Examples are Q-learning, which learns a value function, actor-critic algorithms, which learn a value function and the policy that is greedy with respect to it, and TD(1), which learns a value function based on future rewards. Other algorithms are pure *policy-search* algorithms; they directly learn a policy that returns high rewards. These include REINFORCE (Williams, 1987, Williams, 1987b, Williams, 1988), backpropagation through time, learning automata, and genetic algorithms. The algorithms proposed here combine the two approaches: they perform *Value And Policy Search* (VAPS). The general VAPS equation is instantiated by choosing an expression for e. This can be a Bellman residual (yielding value-based), the reinforcement (yielding policy-search), or a linear combination of the two (yielding Value And Policy Search). The single VAPS update rule on the left side of Table 6.2 generates a variety of different types of algorithms, some of which are described in the following sections.

6.3.1 Reducing Mean Squared Residual Per Trial

If the MDP has terminal states, and a *trial* is the time from the start until a terminal state is reached, then it is possible to minimize the expected total error per trial by resetting the trace to zero at the start of each trial. Then, a convergent form of SARSA, Q-learning, incremental value iteration, or advantage learning can be generated by choosing e to be the squared Bellman residual, as shown on the right side of Table 6.2. In each case, the expected value is taken over all possible (x_t,u_t,R_t) triplets, given s_{t-1}. The policy must be a smooth, nonzero function of the weights. So it could not be an ε-greedy policy that chooses the greedy action with probability (1-ε) and chooses uniformly otherwise. That would cause a discontinuity in the gradient when two Q values in a state were equal. However, the policy could be something that approaches ε-greedy as a positive temperature c approaches zero:

$$P(u \mid x) = \frac{\varepsilon}{n} + (1 - \varepsilon)\frac{1 + e^{Q(x,u)/c}}{\sum_{u'} \left(1 + e^{Q(x,u')/c}\right)} \tag{6.4}$$

where n is the number of possible actions in each state. Note that this is just an example, not part of the definition of the VAPS algorithm. VAPS is designed to work with any smooth function approximator and any smooth exploration policy. This particular exploration policy was used in the simulations shown here, but any other smooth function could have been used instead.

For each instance in Table 6.2 other than value iteration, the gradient of e can be estimated using two, independent, unbiased estimates of the expected value. For example:

$$\frac{\partial}{\partial w} e_{SARSA}(s_t) = e_{SARSA}(s_t)\left(\gamma\phi \frac{\partial}{\partial w} Q(x'_t, u'_t) - \frac{\partial}{\partial w} Q(x_{t-1}, u_{t-1})\right) \tag{6.5}$$

When ϕ=1, this is an estimate of the true gradient. When ϕ<1, this is a *residual* algorithm, as described in chapter 4, and it retains guaranteed convergence, but may learn more quickly than pure gradient descent for some values of ϕ. Note that the gradient of $Q(x,u)$ at time t uses primed variables. That means a new state and action at time t were generated independently conditioned on the state and action at time t-1. Of course, if the MDP is deterministic, then the primed variables are the same as the unprimed. If the MDP is nondeterministic but the model is known, then the model must be evaluated one additional time to get the other state. If the model is not known, then there are three choices. First, a model could be learned from past data, and then evaluated to give this independent sample. Second, the issue could be ignored, simply reusing the unprimed variables in place of the primed variables. This may affect the quality of the learned function (depending on how random the MDP is), but doesn't stop convergence, and may be an acceptable approximation in practice. In fact, this is recommended for POMDPs. Third, all past transitions could be recorded, and the primed variables could be found by searching for all the times (x_{t-1},u_{t-1}) has been seen before, and randomly choosing one of those transitions and using its successor state and action as the primed variables. This is equivalent to learning the certainty equivalence model, and sampling from it, and so is a special case of the first choice. For extremely large state-action spaces with many starting states, this is likely to give the same result in practice as simply reusing the unprimed variables as the primed variables. Note that when weights do not affect the policy at all, these algorithms reduce to standard residual algorithms.

It is also possible to reduce the mean squared residual per step, rather than per trial. This is done by making period lengths independent of the policy, so minimizing error per period will also minimize the error per step. For example, a period might be defined to be the first 100 steps, after which the traces are reset, and the state is returned to the start state. Note that if every state-action pair has a positive chance of being seen in the first 100 steps, then this will *not* just be solving a finite-horizon problem. It will be actually be solving the discounted, infinite-horizon problem, by reducing the Bellman residual in every state. However, the weighting of the residuals will be determined only by what happens during the first 100 steps. Many different problems can be solved by the VAPS algorithm by instantiating the definition of "period" in different ways. These are not different algorithms for solving the same problem. Rather, they are algorithms for solving different problems, with different metrics. When searching for a good value function, it is clearly good to find one with zero Bellman residual everywhere, but if that is not possible, then it is not clear how best to weight the residuals. The goal might be to reduce average error per trial or average error per step. Either way, it is easy to derive a VAPS algorithm that tries to optimize that criterion.

6.3.2 Policy-Search and Value-Based Learning

It is also possible to add a term that tries to maximize reinforcement directly. For example, e could be defined to be $e_{SARSA\text{-}policy}$ rather than e_{SARSA}. from Table 6.2, and the trace reset to zero after each terminal state is reached. The constant b does not affect the expected gradient, but does affect the noise distribution, as discussed in (Williams, 88). When β=0, the algorithm will try to learn a Q function that satisfies the Bellman equation, just as before. When β=1, it directly learns a policy that will minimize the expected total discounted reinforcement. The resulting "Q function" may not even be close to containing true Q values or to satisfying the Bellman equation, it will just give a good policy. When β is in between, this algorithm tries to both satisfy the Bellman equation and give good greedy policies. A similar modification can be made to any of the algorithms in Table 6.2. In the special case where β=1, this algorithm reduces to the REINFORCE algorithm (Williams, 1988). REINFORCE has been rederived for the special case of gaussian action distributions (Tresp & Hofman, 1995), and extensions of it appear in (Marbach, 1998). This case of pure policy search is particularly interesting, because for β=1, there is no need for any kind of model or of generating two independent successors. Other algorithms have been proposed for finding policies directly, such as those given in (Gullapalli, 92) and the various algorithms from learning automata theory summarized in (Narendra & Thathachar, 89). The VAPS algorithms proposed here appears to be the first one unifying these two approaches to reinforcement learning, finding a value function that both approximates a Bellman-equation solution and directly optimizes the greedy policy.

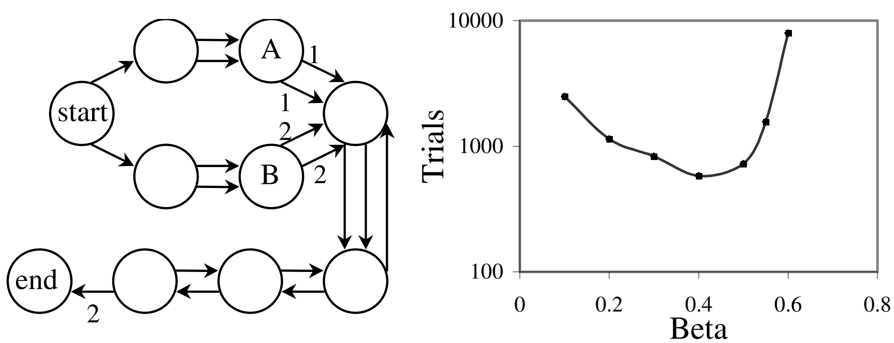

Figure 6.1. A POMDP and the number of trials needed to learn it vs. β. A combination of policy-search and value-based RL outperforms either alone.

Figure 6.1 shows simulation results for the combined algorithm. A run is said to have learned when the greedy policy is optimal for 1000 consecutive trials. The graph shows the average plot of 100 runs, with different initial random weights between $\pm10^{-6}$. The learning rate was optimized separately for each β value. R=1 when leaving state A, R=2 when leaving state *B* or entering *end*, and R=0 otherwise. γ=0.9. The algorithm used was the modified *Q*-learning from Table 6.2, with exploration as in equation 13, and φ=c=1, b=0, ε=0.1. States *A* and *B* share the same parameters, so ordinary SARSA or greedy *Q*-learning could never converge, as shown in (Gordon, 96). When β=0 (pure value-based), the new algorithm converges, but of course it cannot learn the optimal policy in the start state, since those two *Q* values learn to be equal. When β=1 (pure policy-search), learning converges to optimality, but slowly, since there is no value function caching the results in the long sequence of states near the end. By combining the two approaches, the new algorithm learns much more quickly than either alone.

It is interesting that the VAPS algorithms described in the last three sections can be applied directly to a Partially Observable Markov Decision Process (POMDP), where the true state is hidden, and all that is available on each time step is an ambiguous "observation", which is a function of the true state. Normally, an algorithm such as SARSA only has guaranteed convergence when applied to an MDP. The VAPS algorithms will converge in such cases. In fact, simulation results on five particular POMDPs (Peshkin, Meuleau, & Kaelbling, 1999) showed VAPS outperforming SARSA(λ) on three problems (including that from in figure 6.1, with the same function approximator but different exploration policy), and was equally good on two.

6.4 Summary

A new family of algorithms was presented: VAPS. Special cases of it give new algorithms corresponding to *Q*-learning, SARSA, and advantage learning, but with guaranteed convergence for a wider range of problems than was previously possible, including POMDPs. For the first time, these can be guaranteed to converge to a local minimum, even when the exploration policy changes during learning. Other special cases allow new approaches to reinforcement learning, where there is a tradeoff between satisfying the Bellman equation and improving the greedy policy. For one MDP, simulation showed that this combined algorithm learned more quickly than either approach alone. This unified theory, unifying for the first time both value-based and policy-search reinforcement learning, is of theoretical interest, and also was of practical value for the simulations performed. Future research with this unified framework may be able to empirically or analytically address the old question of when it is better to learn value functions and when it is better to learn the policy directly. It may also shed light on the new question, of when it is best to do both at once.

7 Conclusion

Gradient descent is a powerful concept that has been underused in reinforcement learning. By rederiving all of the common algorithms with stochastic gradient-descent techniques, it is possible to guarantee their convergence, and to speed them up in practice. It is also possible to derive entirely new algorithms. It was gradient descent that made Advantage learning possible, which is much better than *Q*-learning for continuous-time problems with small time steps. It even allowed VAPS to be derived, which allows adaptive exploration policies, combines the two main approaches to reinforcement learning into a single algorithm in a natural way, and has shown advantages in practice as well as theory. The techniques proposed here, such as residual algorithms (which are faster than pure gradient descent) and the smoothing function (which allows powerful theoretical results to be derived) are all due to the underlying concept of gradient descent. Because all of these ideas were based on a simple derivation from gradient descent on simple error functions, it is possible to combine them with each other, as was done in VAPS, to apply them to general function approximators, to analyze them, and to implement them on simple hardware.

7.1 Contributions

This thesis has proposed a general, unifying concept for reinforcement learning using function approximators and incremental, online learning. The residual and VAPS families of algorithms include a new counterpart to most of the existing algorithms commonly in use. The Advantage updating algorithm was proposed, though it had many major flaws. The power of residual algorithms was further illustrated by deriving Advantage learning from Advantage updating, removing all of the obvious flaws. The power of VAPS was further illustrated by combining values with policy search (hence the name). The figure 7.1 illustrates how these contributions build on each other, where each piece is supported by one or more other pieces. Each box lists one or more contributions, except for the gray box, which contains prior algorithms that already existed.

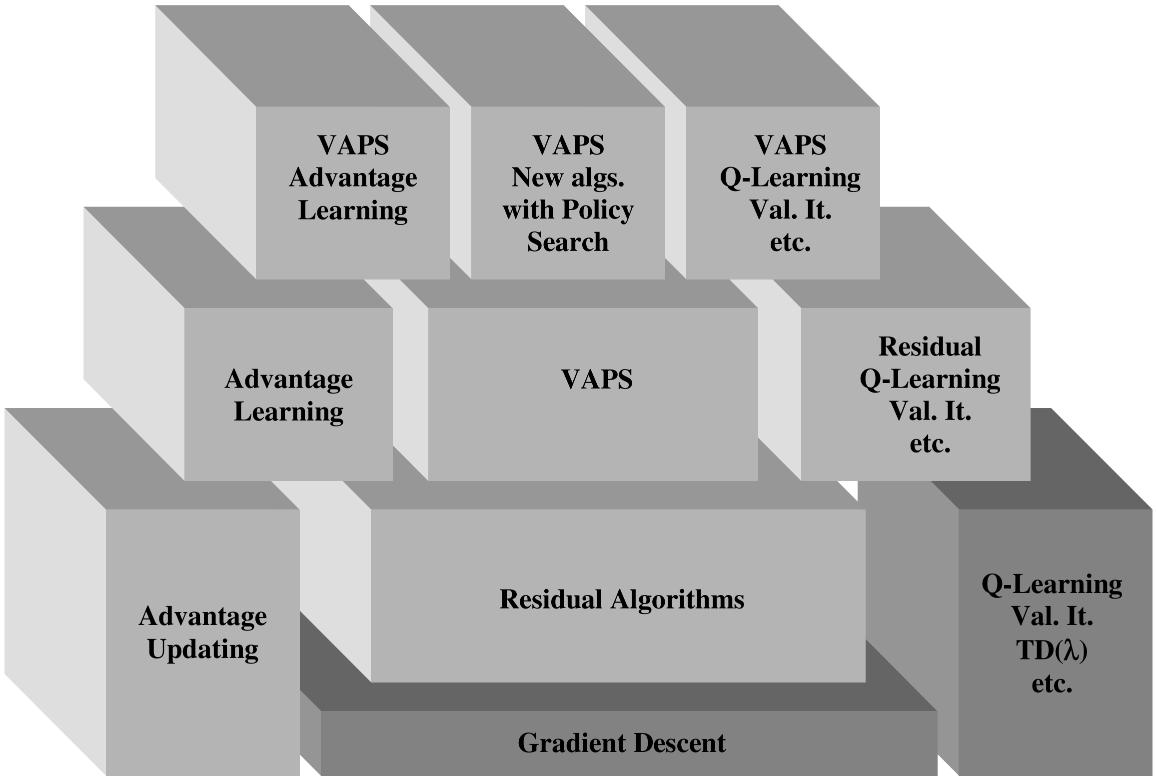

Figure 7.1. Contributions of this thesis (all but the dark boxes), and how each built on one or two previous ones. Everything ultimately is built on gradient descent.

7.2 Future Work

Future work could explore when values are preferable to pure policy search (or when Value and Policy Search together is a better idea). It could further explore whether local minima are a problem in practice. It might examine the application of gradient descent techniques to other forms of reinforcement learning, such as TD(λ) and hierarchical systems. It would be particularly interesting to investigate how policy search as done in VAPS interacts with POMDPs, where values can be a problem. There are many areas in reinforcement learning where gradient descent techniques might be useful, and there is much room for further exploration.

References

Baird, L. C. (1995). Residual Algorithms: Reinforcement Learning with Function Approximation. In Armand Prieditis & Stuart Russell, eds. Machine Learning: Proceedings of the Twelfth International Conference, 9-12 July, Morgan Kaufmann Publishers, San Francisco, CA.

Baird, L. C., & Klopf, A. H. (1993). Reinforcement Learning with High-Dimensional, Continuous Actions. (Technical Report WL-TR-93-1147). Wright-Patterson Air Force Base Ohio: Wright Laboratory.

Baird, L.C. (1993). Advantage updating. Wright-Patterson Air Force Base, OH. (Wright Laboratory Technical Report WL-TR-93-1146, available from the Defense Technical information Center, Cameron Station, Alexandria, VA 22304-6145).

Barto, A., "Connectionist Learning for Control: An Overview," COINS Technical Report 89-89, Department of Computer and Information Science, University of Massachusetts, Amherst, September, 1989.

Barto, A., and S. Singh, "Reinforcement Learning and Dynamic Programming," Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems, New Haven, CT, August, 1990.

Barto, A., R. Sutton, and C. Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems," IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-13, No. 5, September/October 1983.

Barto, A., R. Sutton, and C. Watkins, "Learning and Sequential Decision Making," COINS Technical Report 89-95, Department of Computer and Information Science, University of Massachusetts, Amherst, September, 1989.

Bertsekas, D. P. (1987). Dynamic Programming: Deterministic and Stochastic Models. Englewood Cliffs, NJ: Prentice-Hall.

Bertsekas, D. P. (1995). Dynamic Programming and Optimal Control. Belmont, MA: Athena Scientific.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). Neuro-Dynamic Programming, Belmont, MA: Athena Scientific.

Bertsekas, D. P. and Tsitsiklis, J. N. (1997). Gradient Convergence In Gradient Methods With Errors. MIT Technical Report LIDS-P-2404. (Jan 1999 revision downloadable from http://www.mit.edu/people/dimitrib/Gradient.pdf).

Boyan, J. A., and A. W. Moore. (1995). Generalization in reinforcement learning: Safely approximating the value function. In Tesauro, G., Touretzky, D.S., and Leen, T. K. (eds.), Advances in Neural Information Processing Systems 7. MIT Press, Cambridge MA.

Bradtke, S. J (1993). Reinforcement learning applied to linear quadratic regulation. Proceedings of the Fifth Conference on Neural Information Processing Systems (pp. 295-302). Morgan Kaufmann.

Gaivoronski, A. A. (1994). Convergence properties of backpropagation for neural networks via theory of stochastic gradient methods. Part 1. *Optimization Methods and Software*. 1994.

Gordon, G. (1996). "Stable fitted reinforcement learning". In G. Tesauro, M. Mozer, and M. Hasselmo (eds.), Advances in Neural Information Processing Systems 8, pp. 1052-1058. MIT Press, Cambridge, MA.

Gordon, G. (1999). Approximate Solutions to Markov Decision Processes. Dissertation, Carnegie Mellon University.

Gullapalli, V. (1992). *Reinforcement Learning and Its Application to Control*. Dissertation and COINS Technical Report 92-10, University of Massachusetts, Amherst, MA.

Harmon, M. E., Baird, L. C, & Klopf, A.H. (1995). Advantage updating applied to a differential game. In Tesauro, G., Touretzky, D.S., and Leen, T. K. (eds.), Advances in Neural Information Processing Systems 7. MIT Press, Cambridge MA.

Hornik, K., and M. White, "Multilayer Feedforward Networks are Universal Approximators," Neural Networks, Vol. 2, 1989.

Isaacs, Rufus (1965). Differential games. New York: John Wiley and Sons, Inc.

Kaelbling, L. P., Littman, M. L. & Cassandra, A., “Planning and Acting in Partially Observable Stochastic Domains”. Artificial Intelligence, to appear. Available at http://www.cs.brown.edu/people/lpk.

Mangasarian, O. L. and Solodov, M. V. (1994). "Backpropagation Convergence Via Deterministic Nonmonotone Perturbed Minimization". Advances in Neural Information Processing Systems 6, J. D. Cowan, G. Tesauro, and J. Alspector (eds), Morgan Kaufmann Publisher, San Francisco, CA, 1994.

Marbach, P. (1998). Simulation-Based Optimization of Markov Decision Processes. Thesis LIDS-TH 2429, Massachusetts Institute of Technology.

McCallum (1995), A. *Reinforcement learning with selective perception and hidden state.* Dissertation, Department of Computer Science, University of Rochester, Rochester, NY.

Millington, P. J. (1991). Associative reinforcement learning for optimal control. Unpublished master's thesis, Massachusetts Institute of Technology, Cambridge, MA.

Narendra, K., & Thathachar, M.A.L. (1989). *Learning automata: An introduction*. Prentice Hall, Englewood Cliffs, NJ.

Peshkin, L., Meuleau, N., and Kaelbling, L. (1999). "Learning Policies with External Memory". Submitted to the International Conference on Machine Learning, 1999.

Rajan, N., Prasad, U. R., and Rao, N. J. (1980). Pursuit-evasion of two aircraft in a horizontal plane. Journal of Guidance and Control. 3(3), May-June, 261-267.

Rumelhart, D., G. Hinton, and R. Williams, "Learning Internal Representation by Error Propagation," Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1, Rumelhart, D., and J. McClelland, ed., MIT Press, Cambridge, MA, 1986.

Rumelhart, D., Hinton, G., & Williams, R. (1986). Learning representations by backpropagating errors. Nature. 323, 9 October, 533-536.

Solodov, M. V. (1995). Nonmonotone and Perturbed Optimization. Dissertation, University of Wisconsin – Madison.

Solodov, M. V. (1996). Convergence Analysis of Perturbed Feasible Descent Methods. Journal of Optimization Theory and Applications. **93**(2) pp. 337-353.

Solodov, M. V. and Zavriev, S. K. (1994). Stability Properties of the Gradient Projection Method with Applications to the Backpropagation Algorithm Submitted to SIAM Journal on optimization.

Solodov, M. V. and Zavriev, S. K. (1998). Error Stability Properties of Generalized Gradient-Type Algorithms. Journal of Optimization Theory and Applications. **98**(3) pp. 663-680.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. Machine Learning, 3, 9-44.

Sutton, R., (1990) "Artificial Intelligence by Approximating Dynamic Programming," Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems, New Haven, CT, August.

Tesauro, G. (1990). Neurogammon: A neural-network backgammon program. Proceedings of the International Joint Conference on Neural Networks 3 (pp. 33-40). San Diego, CA.

Tesauro, G. (1992). Practical issues in temporal difference learning. Machine Learning, **8**(3/4), 257-277.

Tresp, V., & R. Hofman (1995). "Missing and noisy data in nonlinear time-series prediction". In *Proceedings of Neural Networks for Signal Processing 5*, F. Girosi, J. Makhoul, E. Manolakos and E. Wilson, eds., IEEE Signal Processing Society, New York, New York, 1995, pp. 1-10.

Tsitsiklis, J., & B. Van Roy (1997). "An analysis of temporal-difference learning with function approximation". IEEE Transactions on Automatic Control. **42**(5), 674-690.

Watkins, C. (1989), "Learning from Delayed Rewards," Ph.D. thesis, Cambridge University, Cambridge, England.

Watkins, C. J. C. H., & Dayan, P. (1992). Technical note: *Q*-learning. Machine Learning, 8(3/4), 279-292.

Weaver, S. E. (1999). A Theoretical Framework for Local Adaptive Networks in Static and Dynamic Systems. Dissertation, University of Cincinnati.

White, H. (1989). Some asymptotic results for learning in single hidden-layer feedforward network models. *Journal of the American Statistical Association* **84**(408) pp. 1003-1013.

Williams, R. and L. Baird (1990), "A Mathematical Analysis of Actor-Critic Architectures for Learning Optimal Controls Through Incremental Dynamic Programming," Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems, New Haven, CT, August, 1990.

Williams, R. J. (1987). "A Class of Gradient-Estimating Algorithms for Reinforcement Learning in Neural Networks. Proceedings of the IEEE First Annual International Conference on Neural Networks, June 1987.

Williams, R. J. (1987b). "Reinforcement-Learning Connectionist Systems". Northeastern University Technical Report NU-CCS-87-3, February.

Williams, R. J. (1988). *Toward a theory of reinforcement-learning connectionist systems*. Technical report NU-CCS-88-3, Northeastern University, Boston, MA.

Williams, R. J., and Baird, L. C. (1993). Tight Performance Bounds on Greedy Policies Based on Imperfect Value Functions. Northeastern University Technical Report NU-CCS-93-14, November.