Chapter 2

Learning Evaluation Functions for Sequential Decision Making

Given only a simulator for a complex task and a measure of overall cumulative performance, how can we efficiently build an evaluation function which enables optimal or near-optimal decisions to be made at every choice point? This chapter discusses approaches based on the formalism of Markov decision processes and value functions. After introducing the notation which will be used throughout this dissertation, I give a review of the literature on value function approximation. I then discuss two original approaches, Grow-Support and ROUT, for approximating value functions robustly in certain restricted problem classes.

2.1 Value Function Approximation (VFA)

The optimal *value function* is an evaluation function which encapsulates complete knowledge of the best expected search outcome attainable from each state:

$$V^*(x) = \text{the expected long-term reward starting from } x, \text{ assuming optimal decisions.} \tag{2.1}$$

Such an evaluation function is ideal in that a greedy local search with respect to V^* will always make the globally optimal move. In this section, I formalize the above definition, review the literature on computing $V^*(x)$, and motivate a new class of approximation algorithms for this problem based on *working backwards*.

2.1.1 Markov Decision Processes

Formally, let our search space be represented as a Markov decision process (MDP), defined by

- a finite set of states X, including a set of start states $S \subset X$;
- a finite set of actions A;
- a reward function $R : X \times A \rightarrow \Re$, where $R(x, a)$ is the expected immediate reward (or negative cost) for taking action a in state x; and

- a transition model $P : X \times X \times A \to \Re$, where $P(x'|x,a)$ gives the probability that executing action a in state x will lead to state x'.

An agent in an MDP environment observes its current state x_t, selects an action a_t, and as a result receives a reward r_t and moves probabilistically to another state x_{t+1}. It is assumed that the agent can fully observe its current state at all times; more general partially observable MDP models [Littman 96] are beyond the scope of this dissertation. The basic MDP model is flexible enough to represent AI planning problems, stochastic games (e.g., backgammon) against a fixed opponent, and combinatorial optimization search spaces. With natural extensions, it can also represent continuous stochastic control domains, two-player games, and many other problem formulations [Littman 94, Harmon *et al.* 95, Littman and Szepesvári 96, Mahadevan *et al.* 97].

Decisions in an MDP are represented by a *policy* $\pi : X \to A$, which maps each state to a chosen action (or, more generally, a probability distribution over actions). I assume that the policy is *stationary*, that is, unchanging over the course of a simulation. For any stationary policy π, the *policy value function* $V^\pi(x)$ is defined as the expected long-term reward accumulated by starting from state x and following policy π thereafter:

$$V^\pi(x) = \mathrm{E}\Big\{\sum_{t=0}^{\infty} \gamma^t R(x_t, \pi(x_t)) \mid x_0 = x\Big\}. \tag{2.2}$$

Here, $\gamma \in [0,1]$ is a *discount factor* which determines the extent of our preference for short-term rewards over long-term rewards. Assuming bounded rewards, V^π is certainly well-defined for any choice of $\gamma < 1$; in the *undiscounted* case of $\gamma = 1$, V^π remains well-defined under the additional condition that every trajectory is guaranteed to terminate, i.e., reach a special *absorbing state* for which all further rewards are 0. Most of the problems considered in this dissertation have this property naturally; furthermore, an arbitrary MDP evaluated with a discount factor $\gamma < 1$ may be transformed into an absorbing MDP whose undiscounted returns are equivalent to the original problem's discounted returns, simply by introducing a termination probability of $1-\gamma$ at each state. Therefore, I will generally assume $\gamma = 1$ throughout this dissertation, giving equal weight to short-term and long-term rewards.

The policy value function V^π satisfies this linear system of *Bellman equations for prediction*:

$$\forall x, \quad V^\pi(x) = R(x, \pi(x)) + \gamma \sum_{x' \in X} P(x'|x, \pi(x)) V^\pi(x') \tag{2.3}$$

The solution to an MDP is an *optimal policy* π^* which simultaneously maximizes $V^\pi(x)$ at every state x. A deterministic optimal policy exists for every MDP [Bellman 57]. The policy value function of π^* is the optimal value function V^* of Equation 2.1. It satisfies the *Bellman equations for control*:

$$\forall x, \;\; V^*(x) = \max_{a \in A} \big[R(x,a) + \gamma \sum_{x' \in X} P(x'|x,a) V^*(x') \big] \tag{2.4}$$

From the value function V^*, it is easy to recover the optimal policy: at any state x, any action which instantiates the max in Equation 2.4 is an optimal choice [Bellman 57]. This formalizes the notion that V^* is an ideal evaluation function.

Algorithms for computing V^* are well understood in the case where the MDP state space is relatively small (say, fewer than 10^7 discrete states), so that V^* can be implemented as a lookup table. In small MDPs, if we have explicit knowledge of the transition model $P(x'|x,a)$ and reward function $R(x,a)$, then V^* may be calculated exactly by a variety of classical algorithms such as *linear programming* [D'Epenoux 63], *policy iteration* [Howard 60], *modified policy iteration* [Puterman and Shin 78], or *value iteration* [Bellman 57]. In small MDPs where the transition model is not explicitly available, we must build V^* from sample trajectories generated by direct interaction with a simulation of the process; in this case, reinforcement learning (RL) methods apply. RL methods are either *model-based*, which means they build an empirical transition model from the sample trajectories and then apply one of the aforementioned classical algorithms (e.g., Dyna-Q [Sutton 90], Prioritized Sweeping [Moore and Atkeson 93])—or *model-free*, which means they estimate V^* values directly (e.g., TD(λ) [Sutton 88], Q-learning [Watkins 89], SARSA [Rummery and Niranjan 94, Singh and Sutton 96]). I will have more to say on the issue of model-based versus model-free algorithms in Section 6.1.2. Broadly speaking, all these algorithms may be viewed as applying dynamic programming in an asynchronous, incremental way; and under suitable conditions, all can still be shown to converge to the exact optimal value function [Bertsekas and Tsitsiklis 96, Singh *et al.* 98].

The situation is very different for practical large-scale decision tasks. These tasks have high-dimensional state spaces, so enumerating V^* in a table is intractable—a problem known as the "curse of dimensionality" [Bellman 57]. Computing V^* requires generalization. One natural approach is to encode the states as real-valued feature vectors and to use a function approximator to fit V^* over the feature space. This approach goes by the name *value function approximation* (VFA) [Boyan *et al.* 95].

2.1.2 VFA Literature Review

The current state of the art in value function approximation is surveyed thoroughly in the book *Neuro-Dynamic Programming* [Bertsekas and Tsitsiklis 96]. Here, I briefly review the history of the field and the state of the art, so as to place this chapter's algorithms in context.

Any review of the literature on reinforcement learning and evaluation functions must begin with the pioneering work of Arthur Samuel on the game of checkers [Samuel 59, Samuel 67]. Samuel implicitly recognized the worth of the value function, saying that

> ... we are attempting to make the score, calculated for the current board position, look like that calculated for the terminal board position of the chain of moves which most probably will occur during actual play. Of course, if one could develop a perfect system of this sort it would be the equivalent of always looking ahead to the end of the game. [Samuel 59, p. 219]

Samuel's program incrementally changed the coefficients of an evaluation polynomial so as to make each visited state's value closer to the value obtained from lookahead search.

In the dynamic-programming community, Bellman [63] and others explored polynomial and spline fits for value function approximation in continuous MDPs; reviews of these efforts may be found in [Johnson *et al.* 93, Rust 96]. But Artificial Intelligence research into evaluation function learning was sporadic until the 1980s. In the domain of chess, Christensen [86] tried replacing Samuel's coefficient-tweaking procedure with least-squares regression, and was able to learn reasonable weights for the chess material-advantage function. In Othello, Lee and Mahajan [88] trained a nonlinear evaluation function on expertly played games, and it played at a high level. Christensen and Korf [86] put forth a unified theory of heuristic evaluation functions, advocating the principles of "outcome determination" and "move invariance"; these correspond precisely to the two key properties of MDP value functions, that they represent long-term predictions and that they satisfy the Bellman equations. Finally, in the late 1980s, the reinforcement learning community elaborated the deep connection between AI search and dynamic programming [Barto *et al.* 89, Watkins 89, Sutton 90, Barto *et al.* 95]. This connection had been unexplored despite the publication of an AI textbook by Richard Bellman himself [Bellman 78].

Reinforcement learning's most celebrated success has also been in a game domain: backgammon [Tesauro 92, Tesauro 94]. Tesauro modified Sutton's TD(λ) algorithm

[Sutton 88], which is designed to approximate V^π for a fixed policy π, to the task of learning an optimal value function V^* and optimal policy. The modification is simple: instead of generating sample trajectories by simulating a fixed policy π, generate sample trajectories by simulating the policy μ which is greedy with respect to the current value function approximation $\tilde{V}$:

$$\mu(x) = \operatorname*{argmax}_{a \in A} \big[R(x,a) + \gamma \sum_{x' \in X} P(x'|x,a) \tilde{V}(x') \big] \tag{2.5}$$

(Occasional non-greedy "exploration" moves are also usually performed [Thrun 92, Singh *et al.* 98], but were found unnecessary in backgammon because of the domain's inherent stochasticity [Tesauro 92].) The modified algorithm has been termed *optimistic TD(λ)* [Bertsekas and Tsitsiklis 96], because little is known of its convergence properties. An implementation is sketched in Table 2.1.2. When $\lambda = 0$, the algorithm strongly resembles Real-Time Dynamic Programming (RTDP) [Barto *et al.* 95], except that RTDP assigns target values at each state by a "full backup" (averaging over all possible outcomes, as in value iteration) rather than TD(0)'s "sample backups" (learning from only the single observed outcome). Applying optimistic TD(λ) with a multi-layer perceptron function approximator, Tesauro's program learned an evaluation function which produced expert-level backgammon play. These results have been replicated by myself [Boyan 92] and others.

Tesauro's combination of optimistic TD(λ) and neural networks has been applied to other domains, including elevator control [Crites and Barto 96] and job-shop scheduling [Zhang and Dietterich 95]. (I will discuss the scheduling application in detail in Section 7.2.) Nevertheless, it is important to note that when function approximators are used, optimistic TD(λ) provides no guarantees of optimality. The following paragraphs summarize the current convergence results for value function approximation. For both the prediction learning (approximating V^π) and control learning (approximating V^*) tasks, the relevant questions are (1) do the available algorithms converge, and (2) if so, how good are their resulting approximations?

We first consider the case of approximating the policy value function V^π of a *fixed* policy π. The TD(λ) family of algorithms applies here. When $\lambda = 1$, TD(λ) reduces to performing stochastic gradient descent to minimize the squared difference between the approximated predictions $\tilde{V}^\pi$ and the observed simulation outcomes. Under standard conditions, using any parametric function approximator, this will converge to a local optimum of the squared-error function. For sufficiently small λ, however, TD(λ) may diverge when nonlinear function approximators are used [Bertsekas and Tsitsiklis 96]. Only in the case where the function approximator is a *linear architecture* over state features has TD(λ) been proven to converge for arbitrary λ [Tsitsiklis and Roy 96].

Optimistic TD($\boldsymbol{\lambda}$) for value function approximation:

Given:

- a *simulation model* for MDP X;
- a function approximator $\tilde{V}(x)$ parametrized by weight vector $\mathbf{w}$;
- a sequence of *step sizes* $\alpha_1, \alpha_2, \ldots$ for incremental weight updating; and
- a parameter $\lambda \in [0,1]$.

Output: a weight vector $\mathbf{w}$ such that $\tilde{V}(x) \approx V^*(x)$.

Set $\mathbf{w} := \mathbf{0}$ *(or an arbitrary initial estimate).*

for $n := 1, 2, \ldots$ **do**: {

1. Using the greedy policy for the current evaluation function $\tilde{V}$ (see Eq. 2.5), generate a trajectory from a start state in X to termination: $x_0 \to x_1 \cdots \to x_T$. Record the rewards $r_0, r_1, \ldots r_T$ received at each step.
2. Update the weights of $\tilde{V}$ from the trajectory as follows:
   for i := T **downto** 0, **do**: {

   $$\mathrm{targ}_i := \begin{cases} r_T \text{ (the terminal reward)} & \text{if } i = T \\ r_i + \lambda \cdot \mathrm{targ}_{i+1} + (1-\lambda) \cdot \tilde{V}(x_{i+1}) & \text{otherwise.} \end{cases}$$

   Update $\tilde{V}$'s weights by delta rule: $\mathbf{w} := \mathbf{w} + \alpha_n(\mathrm{targ}_i - \tilde{V}(x_i))\nabla_w \tilde{V}(x_i)$.

   }

}

TABLE 2.1. Optimistic TD(λ) for undiscounted value function approximation in an absorbing MDP. This easy-to-implement version performs updates after the termination of each trajectory. For an incremental version that performs updates after each transition, refer to [Sutton 87]. In practice, trajectories are often generated using a mixture of greedy moves and exploration moves [Thrun 92, Singh *et al.* 98].

A useful error bound has also been shown in the linear case: the resulting fit is worse than the best possible linear fit by a factor of at most $(1-\gamma\lambda)/(1-\gamma)$, assuming a discount factor of $\gamma < 1$ [Tsitsiklis and Roy 96]. This implies that TD(1) is guaranteed to produce the best fit, but the bound quickly deteriorates as λ decreases. The same qualitative conclusion applies (though the formula for the bound is more complex) for $\gamma = 1$ [Bertsekas and Tsitsiklis 96].

We now proceed to the harder problem of approximating the optimal value function V^*. First, independent of how we construct it, is an approximate value function $\tilde{V}$ useful for deriving a decision-making policy? Singh and Yee [94] show that if $\tilde{V}$ differs from V^* by at most ϵ at any state, then the expected return of the greedy policy for $\tilde{V}$ will be worse than that of the optimal policy by a factor of at most $2\gamma\epsilon/(1-\gamma)$. A similar result holds in the undiscounted case, assuming all policies are proper (γ is then replaced by a contraction factor in a suitably weighted max norm). This bound is not particularly comforting, since $1/(1-\gamma)$ will be large in practical applications, but at least it guarantees that policies cannot be *arbitrarily* bad.

How should we construct V^*? In general, algorithms based on value iteration's one-step-backup operator, such as optimistic TD(λ), use function approximator predictions to assign new training values for that same function approximator—a recursive process that may propagate and enlarge approximation errors, leading to parameter divergence. I have demonstrated empirically that such divergence can indeed happen when offline value iteration is combined with commonly used function approximators, such as polynomial regression and neural networks [Boyan and Moore 95]. Small illustrative examples of divergence have also been demonstrated [Baird 95, Gordon 95]. Sutton has argued that certain of these instabilities may be prevented by sampling states along simulated trajectories, as optimistic TD(λ) does [Sutton 96]; but there are no convergence proofs of this as yet.

Parameter divergence in offline value iteration can provably be prevented by using function approximators belonging to the class of *averagers*, such as k-nearest-neighbor [Gordon 95]. However, this class excludes practical function approximators which extrapolate trends beyond their training data (e.g., global or local polynomial regression, neural networks). *Residual algorithms*, which attempt to blend optimistic TD(λ) with a direct minimization of the residual approximation errors in the Bellman equation, are guaranteed stable with arbitrary parametric function approximators [Baird 95]; these methods are promising but as yet unproven on real-world problems.

2.1.3 Working Backwards

Value iteration (VI) computes V^* by repeatedly sweeping over the state space, applying Equation 2.4 as an assignment statement (this is called a "one-step backup") at each state in parallel. Suppose the lookup table is initialized with all 0's. Then after the i^{th} sweep of VI, the table will store the maximum expected return of a path of length i from each state. For so-called *stochastic shortest path problems* in which every trajectory produced by the optimal policy inevitably terminates in an absorbing state [Bertsekas and Tsitsiklis 96], this corresponds to the intuition that VI works by propagating correct V^* values backwards, by one step per iteration, from the terminal states.

I have explored the efficiency and robustness gains possible when VI is modified to take advantage of the working-backwards intuition. There are two main classes of MDPs for which correct V^* values can be assigned by working strictly backwards from terminal states:

1. *deterministic* domains with no positive-reward cycles and with every state able to reach at least one terminal state. This class includes shortest-path and minimum "cost-to-go" problems [Bertsekas and Tsitsiklis 96].

2. (possibly stochastic) *acyclic* domains: domains where no trajectory can pass through the same state twice. Many problems naturally have this property (e.g., board-filling games like tic-tac-toe and Connect-Four, industrial scheduling as described in Section 2.3.4 below, and any finite-horizon problem for which time is a component of the state).

Using VI to solve MDPs belonging to either of these special classes can be quite inefficient, since VI performs backups over the entire space, whereas the only backups useful for improving V^* are those on the "frontier" between already-correct and not-yet-correct V^* values. In fact, for small problems there are classical algorithms for both problem classes which compute V^* more efficiently by explicitly working backwards: for the deterministic class, Dijkstra's shortest-path algorithm; and for the acyclic class, Directed-Acyclic-Graph-Shortest-Paths (DAG-SP) [Cormen *et al.* 90].[1] DAG-SP first topologically sorts the MDP, producing a linear ordering of the states in which every state x precedes all states reachable from x. Then, it runs through that list in reverse, performing one backup per state. Worst-case bounds for VI, Dijkstra, and DAG-SP in deterministic domains with X states and A actions per state are $O(AX^2)$, $O(AX \log X)$, and $O(AX)$, respectively.

[1] Although Cormen et al. [90] present DAG-SP only for deterministic acyclic problems, it applies straightforwardly to the stochastic case.

Another difference between VI and working backwards is that VI repeatedly re-estimates the values at every state, using old predictions to generate new training values. By contrast, Dijkstra and DAG-SP are always explicitly aware of which states have their V^* values already known, and can hold those values fixed. This distinction is important in the context of generalization and the possibility of approximation error.

In sum, I have presented two reasons why working strictly backwards may be desirable: efficiency, because updates need only be done on the "frontier" rather than all over state space; and robustness, because correct V^* values, once assigned, need never again be changed. I have therefore investigated generalizations of the Dijkstra and DAG-SP algorithms specifically modified to accommodate huge state spaces and value function approximation. My variant of Dijkstra's algorithm, called Grow-Support, was presented in [Boyan and Moore 95] and is summarized briefly in Section 2.2. My variant of DAG-SP is an algorithm called ROUT [Boyan and Moore 96], which I describe in more detail and with new results in Section 2.3. Other researchers have also investigated learning control by working backwards, notably Atkeson [94] for the case of deterministic domains with continuous dynamics.

2.2 VFA in Deterministic Domains: "Grow-Support"

This section summarizes Grow-Support, an algorithm for value function approximation in large, deterministic, minimum-cost-to-goal domains [Boyan and Moore 95]. Grow-Support is designed to construct the optimal value function with a generalizing function approximator while remaining robust and stable. It recognizes that function approximators cannot always be relied upon to fit the intermediate value functions produced by value iteration. Instead, it assumes only that the function approximator can represent the final V^* function accurately, if given accurate training values for a prespecified collection of sample states. The specific principles of Grow-Support are as follows:

1. We maintain a "support" subset of sample states whose final V^* values have been computed, starting with terminal states and then growing backward from there. The fitter $\tilde{V}$ is trained only on these values, which we assume it is capable of fitting.

2. Instead of propagating values by one-step backups, we use *rollouts*—simulated trajectories guided by the current greedy policy on $\tilde{V}$. They explicitly verify the achievability of a state's estimated future reward before that state is added to the support set. In a rollout, the new $\tilde{V}$ training value is derived from rewards

along an actual path to the goal, not from the predictions made by the previous iteration's function approximation. This prevents divergence.

3. We take maximum advantage of generalization. On each iteration, we add to the support set any sample state that can, by executing a single action, reach a state that passes the rollout test. In a discrete environment, this would cause the support set to expand in one-step concentric "shells" back from the goal, similar to Dijkstra's algorithm. But in the continuous case, the function approximator may be able to extrapolate correctly well beyond the support region—and when this happens, we can add many points to the support set at once. This leads to the very desirable behavior that the support set grows in big jumps in regions where the value function is smooth.

Grow-Support($\hat{X}, G, A,$ NextState, Cost, $\tilde{V}$):

Given: • a finite collection of states $\hat{X} = \{x_1, x_2, \ldots x_N\}$ sampled from the continuous state space $X \subset \Re^n$, and goal region $G \subset X$
- a finite set of allowable actions A
- a deterministic transition function NextState : $X \times A \rightarrow X$
- the 1-step cost function Cost : $X \times A \rightarrow \Re$
- a smoothing function approximator $\tilde{V}$
- a tolerance level ϵ for value function approximation error

Support := $\{(x_i \mapsto 0) \mid x_i \in G\}$
repeat
  Train $\tilde{V}$ to approximate the training set Support
  for each $x_i \notin$ Support do
    $c := \min_{a \in A} \left[\text{Cost}(x_i, a) + \text{RolloutCost}(\text{NextState}(x_i, a), \tilde{V})\right]$
    if $c < \infty$ then
      add $(x_i \mapsto c)$ to the training set Support
until Support stops growing or includes all sample points.

RolloutCost(state x, fitter $\tilde{V}$):
  Starting from x, simulate the greedy policy defined by value function $\tilde{V}$ until either reaching the goal, or exceeding a total path cost of $\tilde{V}(x) + \epsilon$.
  Then return:
  $\longrightarrow$ the actual total cost of the path, if goal is reached with cost $\leq \tilde{V}(x) + \epsilon$;
  $\longrightarrow \infty$, if goal is not reached in cost $\tilde{V}(x) + \epsilon$.

Table 2.2. The Grow-Support algorithm and RolloutCost subroutine

The algorithm is sketched in Table 2.2. In a series of experiments reported in [Boyan and Moore 95], I found that Grow-Support is more robust than value iteration with function approximation. (Several follow-up studies provide additional insight into value iteration's potential for divergence [Gordon 95, Sutton 96].) Grow-Support was also seen to be no more computationally expensive, and often much cheaper, despite the overhead of performing rollouts. Reasons for this include: (1) the rollout test is not expensive; (2) once a state has been added to the support, its value is fixed and it needs no more computation; and most importantly, (3) the aggressive exploitation of generalization enables the algorithm to converge in very few iterations.

It is easy to prove that Grow-Support will always terminate after a finite number of iterations. If the function approximator is inadequate for representing the V^* function, Grow-Support may terminate before adding all sample states to the support set. When this happens, we then know exactly which of the sample states are having trouble and which have been learned. This suggests potential schemes for adaptively adding sample states in problematic regions. The ROUT algorithm, described next, does adaptively generate its own set of sample states for learning.

2.3 VFA in Acyclic Domains: "ROUT"

As Grow-Support scaled up Dijkstra's algorithm for deterministic domains, ROUT aims to scale up DAG-Shortest-Paths (DAG-SP) for stochastic, acyclic domains. In large combinatorial spaces requiring function approximation, DAG-SP's key preprocessing step—topologically sorting the entire state space—is no longer tractable. Instead, ROUT must expend some extra effort to identify states on the current frontier. Once identified (as described below), a frontier state is assigned its optimal V^* value by a simple one-step backup, and this $\{\text{state}\rightarrow\text{value}\}$ pair is added to a training set for a function approximator. I determine the training value by a one-step backup rather than rollouts because, unlike the deterministic MDPs to which Grow-Support applies, stochastic MDPs would require performing not one but many rollouts for accurate value determination. However, ROUT still does use an analogue of Grow-Support's "rollout test" to identify the states at which the one-step backup may be safely applied.

In sum, ROUT's main loop consists of identifying a frontier state; determining its V^* value; and retraining the approximator. The training set, constructed adaptively, grows backwards from the goal. HuntFrontierState is the key subroutine ROUT uses to identify a good state to add to the training set. The criteria for such a state x are as follows:

1. All states reachable from x should already have their V^* values correctly approximated by the function approximator. This ensures that the policy from x onward is optimal, and that a correct target value for $V^*(x)$ can be assigned.

2. x itself should *not* already have its V^* value correctly approximated. This condition aims to keep the training set as small as possible, by excluding states whose values are correct anyway thanks to good generalization.

3. x should be a state that we care to learn about. For that reason, ROUT considers only states which occur on trajectories emanating from a starting state of the MDP.

The HuntFrontierState operation returns a state which with high probability satisfies these properties. It begins with some state x and generates a number of trajectories from x, each time checking to see whether all states along the trajectory are self-consistent (i.e., satisfy Equation 2.4 to some tolerance ϵ). If all states after x on all sample trajectories are self-consistent, then x is deemed ready, and ROUT will add x to its training set. If, on the other hand, a trajectory from x reveals any inconsistencies in the approximated value function, then we flag that trajectory's *last* such inconsistent state, and restart HuntFrontierState from there. Table 2.3 specifies the algorithm, and Figure 2.3 illustrates how the routine works.

The parameters of the ROUT algorithm are H, the number of trajectories generated to certify a state's readiness, and ϵ, the tolerated Bellman residual. ROUT's convergence to the optimal V^*, assuming the function approximator can fit the V^* training set perfectly, can be guaranteed in the limiting case where $H \to \infty$ (assuring exploration of all states reachable from x) and $\epsilon = 0$. In practice, of course, we want to be tolerant of some approximation error. Typical settings I used were $H = 20$ and $\epsilon =$ roughly 5% of the range of V^*.

The following sections present experimental results with ROUT on three domains: a prediction task, a two-player dice game, and a k-armed bandit problem. For all problems, I compare ROUT's performance with that of optimistic TD(λ) given the equivalent function approximator. I measure the time to reach best performance (in terms of total number of state evaluations performed) and the quality of the learned value function (in terms of Bellman residual, closeness to the true V^*, and performance of the greedy control policy). The results show that ROUT learned evaluation functions which were as good or better than those learned by TD(λ), and used an order of magnitude less training data in doing so. I also report preliminary results on a fourth domain, a simplified production scheduling task.

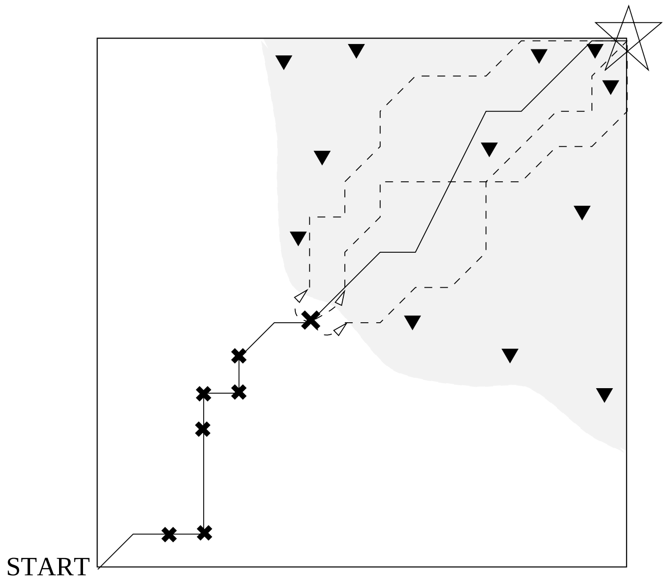

FIGURE 2.1. A schematic of ROUT working on an acyclic two-dimensional navigation domain, where the allowable actions are only $\rightarrow$, $\nearrow$, and $\uparrow$. Suppose that ROUT has thus far established training values for V^* at the triangles, and that the function approximator has successfully generalized V^* throughout the shaded region. Now, when HUNTFRONTIERSTATE generates a trajectory from the start state to termination (solid line), it finds that several states along that trajectory are inconsistent (marked by crosses). The last such cross becomes the new starting point for HUNTFRONTIERSTATE. From there, all trajectories generated (dashed lines) are fully self-consistent, so that state gets added to ROUT's training set. When the function approximator is re-trained, the shaded region of validity should grow, backwards from the goal.

ROUT(start states $\hat{X}$, fitter $\tilde{V}$):
  /* *Assumes that the world model MDP is known and acyclic.* */
  initialize training set $S := \emptyset$, and $\tilde{V} :=$ an arbitrary fit;
  repeat:
    for each start state $x \in \hat{X}$ not yet marked "done", do:
      $s :=$ HuntFrontierState$(x, \tilde{V})$;
      add $\{s \mapsto$ one-step-backup(s)$\}$ to training set S and re-train fitter $\tilde{V}$ on S;
      if $(s = x)$, then mark start state x as "done".
  until all start states in $\hat{X}$ are marked "done".

HuntFrontierState(state x, fitter $\tilde{V}$):
  /* *If the value function is self-consistent on all trajectories from x, return x. (That is determined probabilistically by Monte Carlo trials.) Otherwise, return a state on a trajectory from x for which the self-consistency property* **is** *true.* */
  for each legal action $a \in A(x)$, do:
    repeat up to H times:
      generate a trajectory $\vec{T}$ from x to termination, starting with action a;
      let y be the *last* state on $\vec{T}$ with Bellman residual $> \epsilon$;
      if $(y \neq \emptyset)$ and $(y \neq x)$, then break out of loops, and
        restart procedure with HuntFrontierState$(y, \tilde{V})$.
  /* *reaching this point, x's subtree is deemed all self-consistent and correct!* */
  return x.

Table 2.3. The ROUT algorithm and the HuntFrontierState subroutine

2.3.1 Task 1: Stochastic Path Length Prediction

The "Hopworld" is a small domain designed to illustrate how ROUT combines working backwards, adaptive sampling and function approximation. The domain is an acyclic Markov chain of 13 states in which each state has two equally probable successors: one step to the right or two steps to the right. The transition rewards are such that for each state $V^*(n) = -2n$. Our function approximator $\tilde{V}$ makes predictions by interpolating between values at every fourth state. This is equivalent to using a linear approximator over the four-element feature vector representation depicted in Figure 2.2.

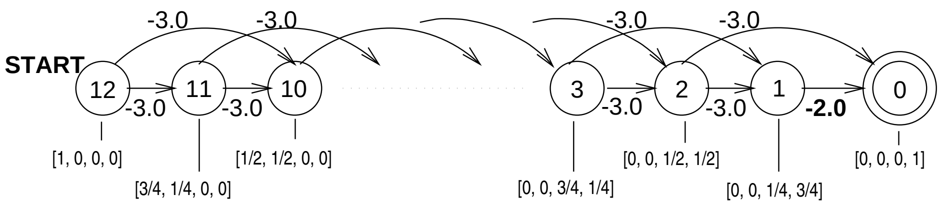

FIGURE 2.2. The Hopworld Markov chain. Each state is represented by a four-element feature vector as shown. The function approximator is linear.

In ROUT, we fit the training set using a batch least-squares fit. In TD, the coefficients are updated using the delta rule with a hand-tuned learning rate. The results are shown in Table 2.4. ROUT's performance is efficient and predictable on this contrived problem. At the start, HUNTFRONTIERSTATE finds $\tilde{V}$ is inconsistent and trains $\tilde{V}(1)$ and $\tilde{V}(2)$ to be -2 and -4, respectively. Linear extrapolation then forces states 3 and 4 to be correct. On the third iteration, $\tilde{V}(5)$ is spotted as inconsistent and added to the training set, and beneficial extrapolation continues. By comparison, TD also has no trouble learning V^*, but requires many more evaluations. This is because TD trains blindly on all transitions, not only the useful ones; and because its updates must be done with a fairly small learning rate, since the domain is stochastic. TD could be improved by an adaptive learning rate, or better yet, by eliminating its learning rates and performing *Least-Squares TD* as described later in Section 6.1.2.

2.3.2 Task 2: A Two-Player Dice Game

"Pig" is a two-player children's dice game. Each player starts with a total score of zero, which is increased on each turn by dice rolling. The first to 100 wins. On her turn, a player accumulates a subtotal by repeatedly rolling a 6-sided die. If at any time she rolls a 1, however, she loses the subtotal and gets only 1 added to her total.

Thus, before each roll, she must decide whether to (a) add her currently accumulated subtotal to her permanent total and pass the turn to the other player; or (b) continue rolling, risking an unlucky 1.

Pig belongs to the class of symmetric, alternating Markov games. This means that the minimax-optimal value function can be formulated as the unique solution to a system of generalized Bellman equations [Littman and Szepesvári 96] similar to Equation 2.4. The state space, with two-player symmetry factored out, has 515,000 positions—large enough to be interesting, but small enough that computing the exact V^* is tractable.

For input to the function approximator, we represent states by their natural 3-dimensional feature representation: X's total, O's total, and X's current subtotal. The approximator is a standard MLP with two hidden units. In ROUT, the network is retrained to convergence (at most 1000 epochs) each time the training set is augmented. Note that this extra cost of ROUT is not reflected in the results table, but for practical applications, a far faster approximator than backpropagation would be used with ROUT.[2]

The Pig results are charted in Table 2.4 and graphed in Figure 2.3. The graph shows the learning curves for the best single trial of each of six classes of runs: TD(0), TD(0.8) and TD(1), with and without exploration. (The vertical axis measures performance in expected points per game against the minimax optimal player, where +1 point is awarded for a win and −1 for a loss.) The best TD run, TD(0) with exploration, required about 30 million evaluations to reach its best performance of about −0.15. By contrast, ROUT completed successfully in under 1 million evaluations, and performed at the significantly higher level of −0.09. ROUT's adaptively generated training set contained only 133 states.

2.3.3 Task 3: Multi-armed Bandit Problem

Our third test problem is to compute the optimal policy for a finite-horizon k-armed bandit [Berry and Fristedt 85]. While an optimal solution in the infinite-horizon case can be found efficiently using Gittins indices, solving the finite-horizon problem is equivalent to solving a large acyclic, stochastic MDP in belief space [Berry and Fristedt 85]. I show results for $k = 3$ arms and a horizon of $n = 25$ pulls, where the resulting MDP has 736,281 states. Solving this MDP by DAG-SP produces the

[2] Unlike TD, which works only with parametric function approximators for which $\nabla_w \tilde{V}(x)$ can be calculated, ROUT can work with arbitrary function approximators, including batch methods such as projection-pursuit and locally weighted regression. For these comparative experiments, however, we used linear or neural network fits for both algorithms.

Problem	Method	training samples	total evals	RMS Bellman	RMS $\|V^*-F\|$	policy quality
HOP	Discrete*	12	21	0	0	-24 *
	ROUT	4	158	0.	0.	-24
	TD(0)	5000	10,000	0.03	0.1	-24
	TD(1)	5000	10,000	0.03	0.1	-24
PIG	Discrete*	515,000	3.6M	0	0	0 *
	ROUT	133	0.8M	0.09	0.14	-0.093
	TD(0) + explore	5 M	30 M	0.23	0.29	-0.151
	TD(0.8) + explore	9 M	60 M	0.23	0.33	-0.228
	TD(1) + explore	6 M	40 M	0.22	0.30	-0.264
	TD(0) no explore	8+ M	50+ M	0.12	0.54	-0.717
	TD(0.8) no explore	5 M	35 M	0.33	0.44	-0.308
	TD(1) no explore	5 M	30 M	0.23	0.32	-0.186
BAND	Discrete*	736,281	4 M	0	0	0.682 *
	ROUT	30	15,850	0.01	0.05	0.668
	TD(0)	150,000	900,000	0.07	0.14	0.666
	TD(1)	100,000	600,000	0.02	0.04	0.669

TABLE 2.4. Summary of results. For each algorithm on each problem, I list two measurements of time to quiescence followed by three measurements of the solution quality. The measurements for TD were taken at the time when, roughly, best performance was first consistently reached. (Key: M=10^6; * denotes optimal performance for each task.)

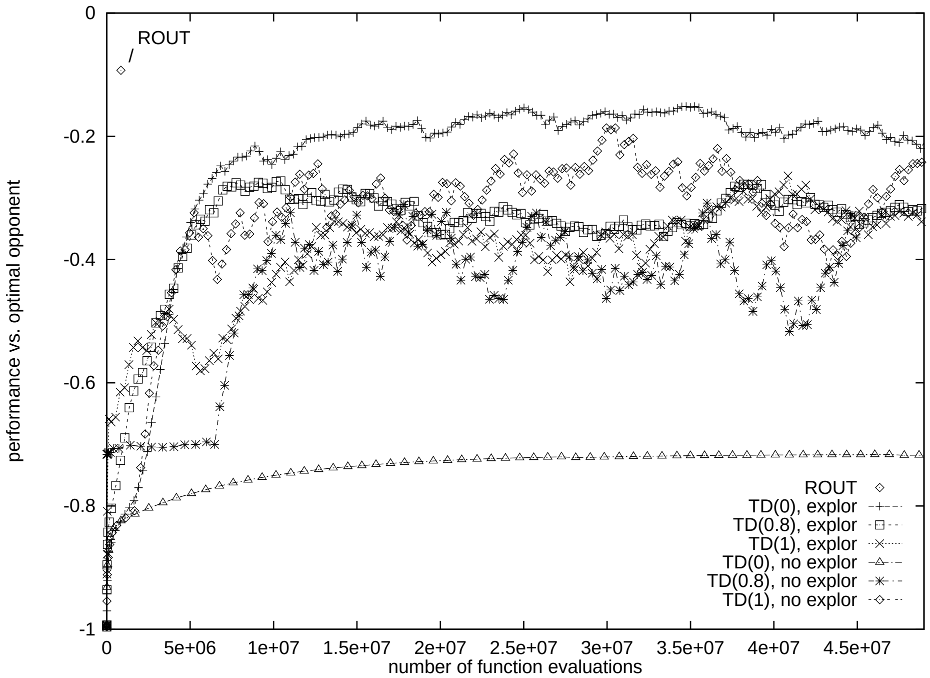

FIGURE 2.3. Performance of Pig policies learned by TD and ROUT. ROUT's performance is marked by a single diamond at the top left of the graph.

optimal exploration policy, which has an expected reward of 0.6821 per pull.

Each state is encoded as a six-dimensional feature vector of [#succ$_{arm1}$, #fail$_{arm1}$, #succ$_{arm2}$, #fail$_{arm2}$, #succ$_{arm3}$, #fail$_{arm3}$] and attempted to learn a neural network approximation to V^* with TD(0), TD(1), and ROUT. Again, the parameters for all algorithms were tuned by hand.

The results are shown in Table 2.4. All methods do spectacularly well, although the TD methods again require more trajectories and more evaluations. Careful inspection of the problem reveals that a globally linear value function, extrapolated from the states close to the end, has low Bellman residual and performs very nearly optimally. Both ROUT and TD successfully exploit this linearity.

2.3.4 Task 4: Scheduling a Factory Production Line

Production scheduling, the problem of deciding how to configure a factory sequentially to meet demands, is a critical problem throughout the manufacturing industry.[3] We assume we have a modest number of products (2–100) and must produce enough of each to keep warehouse stocks high enough to satisfy customer requests for bulk shipments. This production model is common, for example, for most goods found in a supermarket.

An instance of the production scheduling problem is composed of five parts:

Machines and products. This is a list of what machines are present in the factory, and what products can be made on the machines. There may be complex constraints such as "machine A can only make product 1 when machine B is not making product 3." A complete, legal assignment of products onto the set of machines is called a *configuration*. There is also a special "closed" configuration which represents a decision to shut the factory down.

Changeover times. It generally takes a certain amount of time to switch the factory from one configuration to another. During that time, there is no production. The problem definition includes a (possibly stochastic) estimate of how long it takes to change each configuration to each other configuration.

Production rates. Each configuration produces a set of products at a certain rate. There may be dependencies between the machines. For example, machine B may produce product 2 faster if machine A is also producing product 2. The actual production rates in the factory may be very stochastic; for example, some machines may jam frequently, causing irregular delays on the production line.

[3] The application of RL to production scheduling reported here is the result of a collaboration with Jeff Schneider and Andrew Moore [Schneider *et al.* 98].

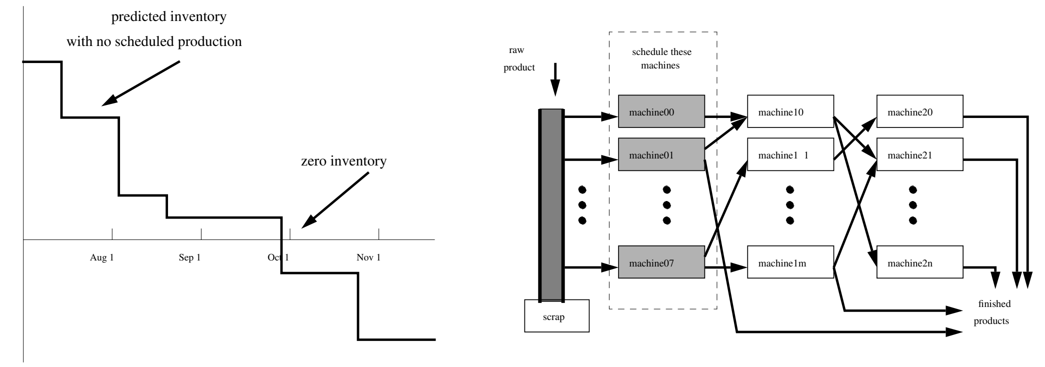

FIGURE 2.4. Demand inventory curves (left) and factory layout (right). See text for further explanation.

Inventory demand curves. At the time a schedule is created, a demand curve for each product is available from a corporate marketing and forecasting group. As shown in Fig. 2.4, each curve starts at the left with the current inventory of that product. The inventory decreases over time as future product shipments are made and eventually goes below zero if no new production occurs. To avoid penalties, the scheduler should call for more production before the demand curve falls below zero. These curves may also change over time as new information about future product demand becomes available.

Schedule costs. Running a schedule generates a dollar measure of net profit or loss. This includes the costs of running the factory, paying the workers, purchasing the raw materials, and carrying inventory at the warehouse. It also includes heuristic costs such as an estimate of the damage done by failing to fill a customer request when the warehouse inventory goes to zero. Finally, it includes the revenue generated from selling product to a customer.

Given this problem description, the task of production scheduling is to maximize expected profit by selecting factory configurations over a period of time. In cases where the production rates and demand curves are assumed deterministic, the problem reduces to finding the optimal *open-loop* schedule: that is, find a fixed sequence of configurations that maximizes profit. In the general stochastic case, the optimal choice of configuration at time t will depend on the outcomes of earlier configurations, so the optimal solution has the form of a *closed-loop* scheduling policy.

The production scheduling problem is modelled very naturally as a Markov Decision Process, as follows:

- The system state is defined by the current time $t \in 0 \ldots T$; the current inventory of each product $p_1 \ldots p_N$; and, if there are configuration-dependent changeover times, the current factory configuration.

- The action set consists of all legal factory configurations. We assume a discrete-time model, so the configuration chosen at time t will run unchanged until time $t+1$.
- The stochastic transition function applies a simulation of the factory to compute the change in all inventory levels realized by running configuration c_t for 1 timestep. This model handles random variations in production rates straightforwardly; it also handles changeover times by simply decreasing production in proportion to the (possibly stochastic) downtime. The time t is incremented on each step, and the process terminates when $t = T$.
- The immediate reward function is computed from the inventory levels, based on the demand curve at time t. It incorporates the revenues from production, penalties from late production, employee costs, operating costs and changeover cost incurred during the period. On the final time period (transition from $t = T - 1$ to T), a terminal "reward" assigns additional penalties for any outstanding unsatisfied demands.

The MDP model fully represents uncertainty in production rates and changeover times. As defined here, the model also handles noise in the demands if that noise is time-independent, but it cannot account for the possibility of the demand curves being randomly updated in the middle of a schedule, since that would make the MDP transition probabilities nonstationary. Finally, since the current time t is included as part of the state, the MDP is acyclic: ROUT may be applied.

I applied ROUT to a highly simplified version of a real factory's scheduling problem. The task involves scheduling 8 weeks of production; however, configurations may be changed only at 2-week intervals, and only 17 configuration choices are available. Of these 17, nine have deterministic production rates; the other eight each have two stochastic outcomes, producing only 1/3 of their usual amount with probability 0.5. With a total of $9 \times 1 + 8 \times 2 = 25$ outcomes possible from every state, and four scheduling periods, there are $25^4 = 390,625$ possible trajectories through the space. The optimal policy can be computed by tabulating $V^*(x)$ at every possible intermediate state x of the factory, of which there are $1 + 25 + 25^2 + 25^3 = 16,276$. The optimal policy results in an expected cumulative reward of $-\$22.8$M. By contrast, a random schedule attains a reward of $-923M$. A greedy policy, which at each step selects a configuration to maximize only the next period's profit, attains $-\$97.9$M.

I applied ROUT to this instance, trying two different memory-based function approximators: 1-nearest neighbor and locally weighted linear regression [Cleveland and Devlin 88]. (The local regression used a kernel width of $\frac{1}{8}$ of the range of each input dimension in the training data; this fraction was tuned manually over powers of $\frac{1}{2}$.) Since these function approximators are nonparametric, TD(λ) cannot be used to train them, so ROUT is compared only to the optimal, greedy, and random policies. ROUT's exploration and tolerance parameters were also tuned manually.

Algorithm	Mean Profit	95% C.I.	optimal runs
Optimal	-22.8		1
Random	-923.2	±58.7	0
Greedy	-97.9	±15.1	0
ROUT + nearest neighbor	N/A		0
ROUT + locally weighted linear	-45.0	±16.9	10/16

Table 2.5. Results on the production scheduling task

Table 2.5 summarizes the results. When nearest-neighbor was used as the function approximator, ROUT did not obtain sufficient generalization from its training set and failed to terminate within a limit of several hours. However, with a locally weighted regression model, ROUT did run to completion and produced an approximate value function which significantly outperformed the greedy policy. Moreover, over half of these runs did indeed terminate with the *optimal* closed-loop scheduling policy. In these cases, ROUT's final self-built training set for value function approximation consisted of only about 110 training points—a substantial reduction over the $16,276$ required for full tabulation of V^*. ROUT's total running time ($\approx$ 1 hour on a 200 MHz Pentium Pro) was roughly half of that required to enumerate V^* manually.

From these preliminary results, I conclude that ROUT does indeed have the potential to approximate V^* extremely well, given a suitable function approximator for the domain. However, since it runs quite slowly on even this simple problem, I believe ROUT will not scale up to practical scheduling instances without further modification.

2.4 Discussion

When a function approximator is capable of fitting V^*, ROUT will, in the limit, find it. However, for ROUT to be efficient, the frontier must grow backward from the goal quickly, and this depends on good extrapolation from the training set. When good extrapolation does not occur, ROUT may become stuck, repeatedly adding points near the goal region and never progressing backwards. Some function approximators may be especially well-suited to ROUT's required extrapolation from accurate training data, and this deserves further exploration. Another promising refinement would be to adapt the tolerance level ϵ, thereby guaranteeing progress at the expense of accuracy.

Grow-Support and ROUT represent first steps toward a new class of algorithms for solving large-scale MDPs. Their primary innovation is that, without falling prey to the curse of dimensionality, they are able to explicitly represent which states are already solved and which are not yet solved. Using this information, they "work backwards," computing accurate V^* values at targeted unsolved states using either function approximator predictions at solved states or Monte Carlo rollouts. Importantly, and unlike the Dijkstra and DAG-SP exact algorithms on which they are based, they grow the solution set for V^* back from the goal without requiring an explicit backward model for the MDP. Only forward simulations are used; this constrains the distribution of visited states to areas that are actually reachable during task execution. By treating solved and unsolved states differently, Grow-Support and ROUT eliminate the possibility of divergence caused by repeated value re-estimation.

This chapter has reviewed the state of the art in reinforcement learning, a field which is grounded solidly in the theory of dynamic programming but provides few guarantees for the practical cases where function approximators, rather than lookup tables, must be used to construct the value function. I have described two novel algorithms for approximating V^* which are guaranteed stable and which perform well in practice. In the remaining chapters of this thesis, I consider the simpler VFA problem of approximating V^π for a fixed policy π, on which TD(λ) with linear function approximators does come with strong convergence guarantees—and I demonstrate a practical way of exploiting these approximations to improve search performance on global optimization tasks.