problems with up to 126 classes, but tests on very large problems with thousands of classes have not yet been performed.

4 Reinforcement Learning

The previous two sections have discussed problems in supervised learning from examples. This section addresses problems of sequential decision making and control that come under the heading of *reinforcement learning.*

Work in reinforcement learning dates back to the earliest days of artificial intelligence when Arthur Samuel developed his famous checkers program (Samuel, 1959). More recently, there have been several important advances in the practice and theory of reinforcement learning. Perhaps the most famous work is Gerry Tesauro's (1992) TD-gammon program, which has learned to play backgammon better than any other computer program and almost as well as the best human players. Two other interesting applications are the work of Zhang and Dietterich (1995) on job shop scheduling and Crites and Barto (1995) on real-time scheduling of passenger elevators.

Kaelbling, Littman, and Moore (1996) have published an excellent survey of reinforcement learning and Mahadevan and Kaelbling (1996) report on a recent NSF-sponsored workshop on the subject. Two new books (Barto & Sutton, 1997; Bertsekas & Tsitsiklis, 1996) describe the newly-developed reinforcement learning algorithms and the theory behind them. I will summarize these developments here.

4.1 An Introduction to Dynamic Programming

The most important insight of the past five years is that reinforcement learning is best analyzed as a form of online, approximate dynamic programming (Barto, Bradtke, & Singh, 1995). I will introduce this insight using the following notation. Consider a robot interacting with an external environment. At each time t, the environment is in some state s_t, and the robot has available some set of actions A. The robot executes an action a_t, which causes the environment to move to a new state s_{t+1}. A convenient way of specifying the desired behavior of the robot is to define an *immediate reward function* $R(s_t, a, s_{t+1})$ that specifies a real-valued reward for this transition from s_t to s_{t+1}. For example, we can assign a positive reward to actions that reach a desired goal location and we can assign a negative reward to undesirable actions such as colliding with walls, people, or other robots. The immediate reward in all other states could be defined to be zero.

Our long-term goal for the robot can then be defined as some function of the immediate rewards it receives. A commonly-used criterion is the *cumulative discounted reward*,

$$\sum_{t=0}^{\infty} \gamma^t R(s_t, a, s_{t+1}),$$

where $0 \leq \gamma < 1$ is a *discount factor* that controls the relative importance of short-term and long-term rewards.

A procedure or rule for choosing each action a given state s is called the *policy* of the robot, and it can be formalized as a function $a = \pi(s)$. The goal of reinforcement learning algorithms is to compute the *optimal policy*, denoted π^*, which maximizes the cumulative discounted reward.

Researchers in dynamic programming (e.g., Bellman, 1957) found it convenient to define a real-valued function $f^\pi(s)$ called the *value function* of policy π. The value function $f^\pi(s)$ gives the expected cumulative discounted reward that will be received by starting in state s and executing policy π. It can be defined recursively by the formula,

$$f^\pi(s) = \sum_{s'} P(s'|s, \pi(s)) \cdot [R(s, \pi(s), s') + \gamma f^\pi(s')], \tag{1}$$

where $P(s'|s, \pi(s))$ is the probability that the next state will be s' given that the current state is s and we take action $\pi(s)$.

Given a policy π, a reward runction R, and the transition probability function P, it is possible to compute the value function f^π by solving the system of linear equations containing one equation of the form of Equation (1) for each possible state s. The system of equations can be solved via standard methods, such as Gaussian Elimination or Gauss-Seidel iteration or it can be solved iteratively by converting the equation into an assignment statement:

$$f^\pi(s) := \sum_{s'} P(s'|s, \pi(s)) \cdot [R(s, \pi(s), s') + \gamma f^\pi(s')]. \tag{2}$$

This assignment statement is called a *simple backup*, because it can be viewed as taking the current estimated value(s) $f^\pi(s')$ and "backing them up" to compute a revised estimate for $f^\pi(s)$. An example is shown in Figure 7. In state s,

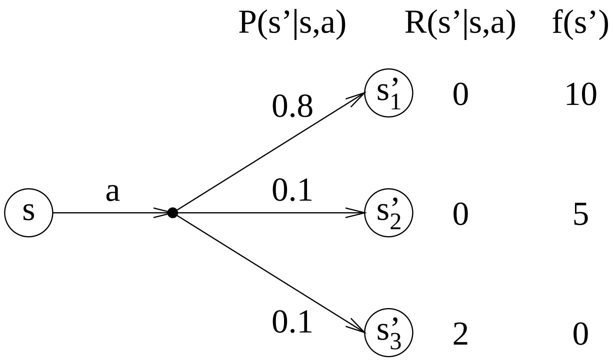

Figure 7: An example of a simple backup. Each arc is labeled with the probability of making that transition. The resulting states are labeled with their associated immediate reward and their value.

Table 6: The Policy Iteration Algorithm

procedure POLICYITERATION(P, R)
let π be an arbitrary initial policy
repeat until π is unchanged
    perform simple backups to compute f^π for each state (Equation 2).
    update π for each state (Equation 3).
    end

we perform action $a = \pi(s)$. There are three possible resulting states, s'_1, s'_2, and s'_3, with probabilities 0.8, 0.1, and 0.1 (respectively). The immediate rewards for reaching these states are 0, 0, and 2 (respectively) and the estimated values $f(s')$ of the states are 10, 5, and 0 (respectively). Assuming $\gamma = 1$, the new estimated value of $f^{\pi(s)}$ is

$$\begin{aligned} f^\pi(s) &:= (0 + 0.8 \cdot 10) + (0 + 0.1 \cdot 5) + (2 + 0.1 \cdot 0) \\ &:= 10.5. \end{aligned}$$

To compute the value of a policy, the simple backups can be performed in any convenient order—even at random. The only requirement is that γ must be small enough so that the sum of the expected discounted rewards converges and the simple backups must be performed until $f^\pi(s)$ converges for every state s.

Given a value function f^π, it is possible to compute a new policy π' that is guaranteed to be at least as good π. This is accomplished by performing a one-step lookahead search and choosing the action whose backed-up value is the largest:

$$\pi'(s) := \operatorname*{argmax}_a \sum_{s'} P(s'|s,a) \cdot [R(s,a,s') + \gamma f^\pi(s')]. \tag{3}$$

In words, we consider each possible action a, compute its expected backed-up value, and define $\pi'(s)$ to be the action that gives the maximum backed-up value.

By alternately computing the value f^π of policy π and then updating π via Equation (3), we can converge to the optimal policy π^* and its optimal value function f^*. Note that if we plug f^* into Equation (3), the new policy is unchanged—it is still the optimal policy. This defines the algorithm known as *policy iteration*, shown in Table 6. It is easy to show that policy iteration converges in a fixed number of iterations. Unfortunately, each iteration can be expensive, because it requires computing the value of the current policy.

An alternative to policy iteration is to work directly with the value function. Bellman proved that the value function f^* of the optimal policy π^* is the unique fixed point of the *Bellman Equation*:

$$f(s) := \max_a \sum_{s'} P(s'|s,a) \cdot [R(s,a,s') + \gamma f(s')]. \tag{4}$$

In other words, we perform a one-step lookahead just as we did for the policy improvement step of policy iteration, but instead of remembering the best action, we update our estimate of the value of state s. This is called a *Bellman Backup*. It is more expensive than a simple backup, because we must consider each possible action a and then each

Table 7: The Value Iteration Algorithm

```
procedure ValueIteration(P, R)
let f be an arbitrary initial value function
repeat until f is unchanged in all states
      for each state s, perform a Bellman backup (Equation 4)
      end
for each state s, compute the optimal policy (Equation 3)
end ValueIteration
```

possible resulting state s'. By performing enough Bellman backups in every state, we can converge to the optimal value function f^*. This is called the *Value Iteration* algorithm, and it is summarized in Table 7.

Unfortunately, value iteration can be more difficult than policy iteration because (a) each backup is a more expensive Bellman backup rather than a simple backup and (b) the value function may take a very long time to converge. Indeed, it is possible for the optimal policy to have converged long before the value function converges.

A hybrid algorithm that combines aspects of both value iteration and policy iteration is called *Modified Policy Iteration*. This algorithm is essentially the same as Policy Iteration except that only a fixed number of simple backups are performed in each state in each iteration. This means that the estimated value function for the current policy f^π does not completely converge to the correct value function, but under fairly mild conditions, it can be shown that the algorithm will still converge to the optimal policy.

All three of these algorithms—policy iteration, value iteration, and modified policy iteration—require performing backups in every state, so their running time scales with the number of states. In fact, value iteration may run for an infinite amount of time without converging. Policy iteration requires time at least $O(n^3)$ for problems with n states just to compute the value of the policy in each iteration. For small problems, this is not a difficulty, but for problems of interest in artificial intelligence, the state space often has 10^{20} or 10^{40} possible states, which renders these algorithms infeasible. Bellman termed this the *curse of dimensionality*, because the number of states, and hence the running time, increases exponentially with the number of dimensions in the state space.

Another drawback of these algorithms is that they require a complete "model" of the system, by which I mean the transition probabilities $P(s'|s,a)$ and the reward function $R(s,a,s')$. There are many applications where this model is unavailable (e.g., in a robot interacting with an unknown environment) or where the model cannot easily be converted into a transition probability matrix. For example, in the elevator control problem studied by Crites and Barto (1995), a software simulator is available that can take a current state s and a proposed action a and generate the next state s' according to the transition probability distribution. However, that distribution is not explicitly represented anywhere, so it is not available for direct use by a dynamic programming algorithm. The problem of constructing an explicit probability transition matrix and reward function has been called the *curse of modeling*, and in many problems it is just as severe than the curse of dimensionality. Reinforcement learning algorithms provide a way of overcoming these two curses.

Reinforcement learning algorithms have introduced three key innovations: (a) stochastic approximation of backups, (b) value function approximation, and (c) model-free learning. I will discuss these innovations in the context of an algorithm known as $TD(\lambda)$ developed by Sutton (1988).

4.2 Temporal Difference Learning and $TD(\lambda)$

I will begin by describing a simplified version of Sutton's $TD(\lambda)$ algorithm, called $TD(0)$. $TD(0)$ is a method for computing approximate simple backups online. Suppose we are in state s and we follow the current policy by taking action $a = \pi(s)$. If we are interacting with a real external environment (or with a simulator), the environment makes a probabilistic transition to a new state s' and produces the immediate reward $R(s,a,s')$. The $TD(0)$ algorithm observes this new state and reward and updates the value function as follows:

$$f^\pi(s) := (1-\alpha)\cdot f^\pi(s) + \alpha \cdot [R(s,a,s') + \gamma f^\pi(s')],$$

where α is a *learning rate parameter*. Typical values for α are between 0.01 and 0.5. (Technically, the α values must shrink to zero over time in order for $TD(0)$ to converge.)

The basic idea of $TD(0)$ is that if we visit s many times and apply action a many times, then by *sampling over time*, we will get the same effect as if we performed a simple backup. We can do this by sampling from the probability distribution $P(s'|s,a)$ rather than by having direct computational access to P.

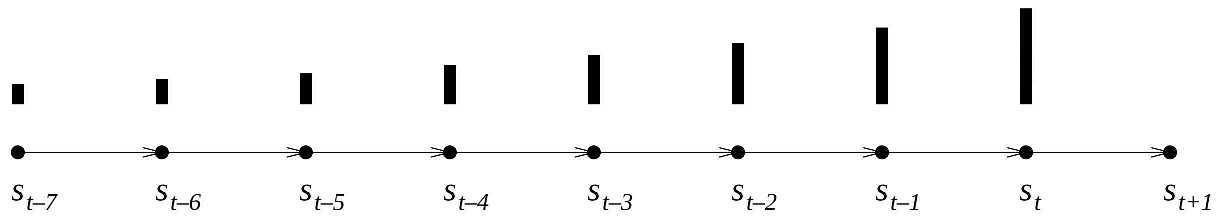

Figure 8: A sequence of states. The eligibility of each state (with $\lambda = 0.8$) is shown as a vertical bar.

This strategy allows $TD(0)$ to compute the value of a policy without having an explicit model. The environment serves as its own model!

When Sutton developed $TD(0)$, he also introduced a second idea. Instead of storing a separate value $f^\pi(s)$ for each state s, suppose we represent the value function as a neural network (or some other differentiable function approximator) of the form $f(s, W)$, where W is a vector of adjustable weights. With this representation, we can't directly assign a value to a state, but we can adjust the weights so that $f(s, W)$ is closer to the desired value. We do this by defining an error function:

$$J(W) = \frac{1}{2}\Big(f^\pi(s, W) - [R(s, a, s') + \gamma f^\pi(s', W)]\Big)^2.$$

This is the squared difference between the current estimated value of state s, $f^\pi(s, W)$, and the backed-up value. This is called the *temporal difference error*. Our goal is to modify W to reduce the temporal difference error $J(W)$. Differentiating $J(W)$ (and treating only the first occurrence of W as adjustable), we obtain the learning rule

$$W := W - \alpha \nabla_W f(s, W)\Big(f^\pi(s, W) - [R(s, a, s') + \gamma f^\pi(s', W)]\Big),$$

where $\nabla_W f(s, W)$ is the gradient of f with respect to the weights W. This takes a step of size α in the direction of the decreasing gradient scaled by the size of the temporal difference error.

By using a smoothly parameterized function approximation $f(s, W)$, $TD(0)$ can circumvent the curse of dimensionality provided that $f(s, W)$ can accurately approximate the true value function $f(s)$ with a small number of parameters W.

Sutton introduced one other wonderful idea in the $TD(\lambda)$ algorithm—the *eligibility trace*. Suppose we have visited a sequence of states s_1, s_2, $\ldots$, s_t, s_{t+1}, and we are updating the value of $f(s_t, W_t)$. Sutton suggested that we might want to update the values of the preceding states as well, since the key idea of dynamic programming is to propagate information about expected rewards backward through the state space. Sutton proposed that we should remember the gradient $\nabla_{W_{t-i}} f(s_{t-i}, W_{t-i})$ for each state s_{t-i}, for $i = 1, \ldots, n$, that we have visited. When we update $f(s_t, W_t)$ by taking a step in the direction of $-\nabla_{W_t} f(s_t, W_t)$, we will also take a smaller step in the direction of $-\nabla_{W_{t-1}} f(s_{t-1}, W_{t-1})$ and an even smaller step in the direction of $-\nabla_{W_{t-2}} f(s_{t-2}, W_{t-2})$ and so on. Each stepsize will be decreased by a factor of $\lambda < 1$. This gives us the learning rule

$$W := W - \alpha \left(\sum_{i=0}^{\infty} \lambda^i \nabla_{W_{t-i}} f(s_{t-i}, W_{t-i})\right) \Big(f^\pi(s_t, W) - [R(s_t, a, s_{t+1}) + \gamma f^\pi(s_{t+1}, W)]\Big).$$

The value of λ^i is called the *eligibility* of state s_{t-i}. Figure 8 shows the eligibility of a sequence of states as a bar graph.

The infinite sum $\sum_{i=0}^{\infty} \lambda^i \nabla_W f(s_{t-i}, W_{t-i})$ can be implemented by maintaining a *current gradient vector* G:

$$G_t := \lambda G_{t-1} + \nabla_{W_t} f(s_t, W_t).$$

With this change, we get the full algorithm $TD(\lambda)$ shown in Table 8. Readers familiar with the momentum method for stabilizing backpropagation will note that the eligibility trace mechanism is very similar. However, in the momentum method, previous *weight changes* are remembered, while in the eligibility trace, previous *gradient vectors* are remembered and future temporal differences determine the stepsize along those previous gradients.

There have been many theoretical studies of the behavior of $TD(\lambda)$. The most general results have been obtained by Tsitsiklis and Van Roy (1996). They analyze the case where the function approximator $f(s, W)$ is a linear

Table 8: The $TD(\lambda)$ algorithm for computing the value of a policy π

---

procedure TD($\pi, \lambda, \gamma, \alpha, s_0$)
initialize $G = \mathbf{0}$
initialize W randomly
s_0 is the starting state
while W has not converged **do**
  take action $a = \pi(s_t)$
  observe resulting state s_{t+1} and reward $R(s_t, a, s_{t+1})$
  $G := \lambda G + \nabla_W f(s_t, W)$
  $W := W - \alpha G \cdot \Big(f(s_t, W) - [R(s_t, a, s_{t+1}) + \gamma f(s_{t+1}, W)]\Big)$
  end while
return W which defines f^π
end TD

---

combination of fixed (and arbitrary) orthogonal basis functions. To analyze how well $f(s, W)$ can approximate f^π, some notion of the *distance* between two value functions is needed. Tsitsiklis and Van Roy define the following measure:

$$||f - f^\pi||_D = \left(\sum_s [f(s) - f^\pi(s)]^2 D(s)\right)^{1/2},$$

where $D(s)$ is the probability that the policy π visits state s.

Using this measure, let $\hat{f}^\pi$ be the best approximation of f^π that can be represented by a linear combination of the given basis functions. Tsitsiklis and Van Roy prove that $TD(\lambda)$ will converge to a function f such that

$$||f - f^\pi||_D \leq \frac{||\hat{f}^\pi - f^*||_D}{1 - \gamma(1-\lambda)/(1-\lambda\gamma)}.$$

The quantity on the left-hand side is the error between the function f learned by $TD(\lambda)$ and the true value function f^π for policy π. The numerator on the right-hand side is the inherent approximation error resulting from the use of a linear combination of the given, fixed basis functions. The denominator is the error that results from the fact that approximation errors in one state will be propagated backward to earlier states. For large λ, the denominator approaches 1 (no error), but for $\lambda = 0$, the denominator becomes $1 - \gamma$, which could be very small, and hence produce very large errors. For this result to hold, it is essential that the backups performed by $TD(\lambda)$ be performed according to the current policy π. If this condition is not observed, then $TD(\lambda)$ may fail to converge.

The $TD(\lambda)$ algorithm provides a way of computing the value of a fixed policy without direct access to the transition probabilities and reward function. However, this is only of limited utility unless we can perform the policy improvement step from Equation (3) and thereby implement the policy iteration algorithm. Unfortunately, policy improvement requires access to a model that can generate the possible next states and their probabilities.

4.3 Applications of $TD(\lambda)$

There are many domains where such a model is available. For example, in game-playing settings, such as backgammon, it is easy to compute the set of available moves from each state and the probabilities of all successor states. Hence, $TD(\lambda)$ can be combined with policy improvement to learn an optimal policy for backgammon. This is what Tesauro did in his famous TD-gammon system (Tesauro, 1992, 1995).

TD-gammon employs a neural network representation of the value of a state. The state of the backgammon game is described by a vector of 198 features that encode the locations of the pieces on the board and the values shown on the dice. TD-gammon begins with a randomly-initialized neural network. It plays a series of games against itself. At each step, it makes a full one-step lookahead search, applies the neural network to evaluate each of the resulting states, and makes the move corresponding to the highest backed up value. In short, it applies Equation (3) to compute the action to perform next. This is a form of local policy improvement. After making the move, it observes the resulting state and applies the $TD(\lambda)$ rule to update the value function. In effect, TD-gammon is executing a form of modified policy iteration where it alternates between one step of policy evaluation (the $TD(\lambda)$ update) and one step of policy improvement (the computation of the best move to make).

There is no guarantee that this algorithm will converge to the optimal policy. Indeed, it is easy to construct examples where the strategy of always performing the best action based on the current approximation to the value

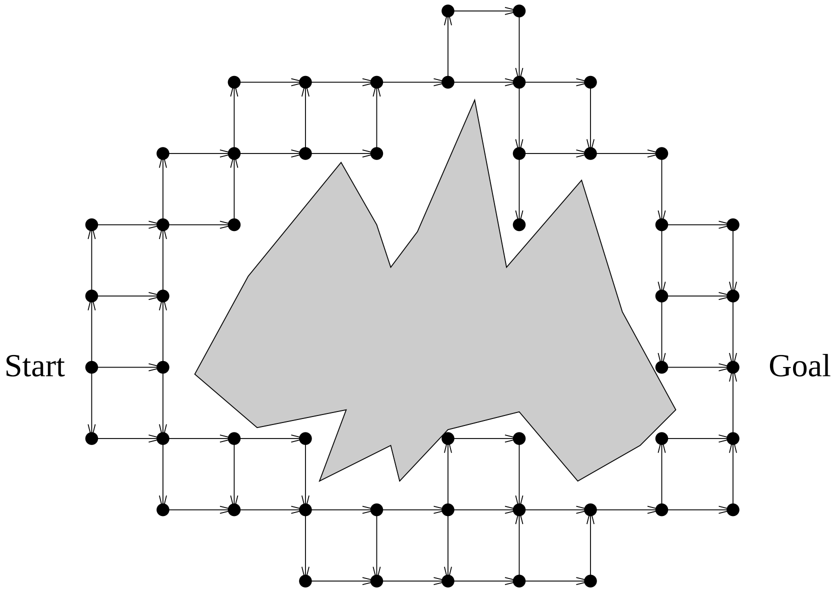

Figure 9: Two networks of roads around a mountain. Without exploration, an initial policy that follows the northern roads will never discover that the southern roads provide a shorter route to the goal.

function, $f(s, W)$, will lead to a local minimum. For example, consider the navigation problem shown in Figure 9. There is a large mountain separating the start state from the goal. A network of roads passes to the north of the mountain, and a similar (and shorter) network passes to the south. Suppose that our initial policy takes us to the north side and we eventually reach the goal. After updating our value function using $TD(\lambda)$, suppose that the north side still appears to be shorter than the south side (because our estimates for the values of the states along the south side are too high). Then, in future trials we will continue to take the north roads, and we will never try the southern route.

This is called the *problem of exploration*. The heart of the problem is that to find the optimal policy, it is necessary to prove that every off-policy action leads to worse expected results than the actions of the optimal policy. In this situation, it is essential to explore the southern path to determine whether it is worse (or better!) than the northern path. The strategy of always taking the action that appears to be optimal based on the current value function is called the *pure exploitation* strategy. The example in Figure 9 shows that the pure exploitation strategy will not always find the optimal policy. Hence, online reinforcement learning algorithms must balance exploitation with exploration.

Fortunately, in backgammon, the random dice rolls inject so much randomness into the game that TD-gammon thoroughly investigates the possible moves in the game. Experimentally, the performance of TD-gammon is outstanding. It plays much better than any other computer program, and it is nearly as good as the world's best players. The results of three versions of the program in three separate matches against human players are shown in Table 9. In some situations, the moves chosen by TD-gammon have been adopted by expert humans.

A similar strategy was applied by Zhang and Dietterich (1995) to the problem of job-shop scheduling. In job-shop scheduling, a set of tasks must be scheduled to avoid resource conflicts. Each task requires certain resources throughout its duration, and each task has prerequisite tasks that must be completed before it can be executed. An optimal schedule is one that completes all of its tasks in the minimum amount of time while satisfying all resource and prerequisite constraints.

Zweben, Daun, and Deale (1994) developed a repair-based search space for this task in which each state is a complete schedule (i.e., all tasks have assigned start times). The starting state is a *critical path* schedule in which every task is scheduled as early as possible subject to its prerequisite constraints and ignoring its resource constraints. The actions in this search space identify the earliest constraint violation and repair it by moving tasks later in time (thus lengthening the schedule). The search terminates when a violation-free schedule is found.

Zhang and Dietterich reformulated this as a reinforcement learning problem where the optimal policy will choose a sequence of repairs that will produce the shortest possible schedule. The immediate reward function gives a small cost to each repair action and a final reward that is inversely proportional to the final length of the schedule. Zhang

Table 9: Summary of the Performance of TD-gammon against some of the world's best players. The results are expressed in net points won (or lost) and in points won per game. Taken from Tesauro (1995).

Program	Training Games	Opponents	Results
TDG 1.0	300,000	Robertie, Davis, Magriel	−13 pts/51 games (−0.25 ppg)
TDG 2.0	800,000	Goulding, Woolsey Snellings, Russell, Sylvester	−7 pts/38 games (−0.18 ppg)
TDG 2.1	1,500,000	Robertie	−1 pts/40 games (−0.02 ppg)

and Dietterich applied $TD(\lambda)$ with a feed-forward neural network to represent the value function. The actions in this domain are deterministic, so deliberate exploration is needed. Their system makes a random exploratory move with a given probability, β, which is gradually decreased during learning. After learning, their system finds schedules that are substantially shorter than the best previous method (for the same expenditure of CPU time). This approach of converting combinatorial optimization problems into reinforcement learning problems should be applicable to many other important industrial domains.

These two applications show that when a simulator is available for a task, it is possible to solve the reinforcement learning problem using $TD(\lambda)$ even in very large search spaces. However, in domains involving interaction with hard-to-model real-world environments (e.g., robot navigation, factory automation), some other method is needed. Two approaches have been explored.

One approach is to learn a predictive model of the environment by interacting with it. Each interaction with the environment provides a training example for supervised learning of the form $s' = env(s,a)$, where env is the environment, s and a are the current state and action, and s' is the resulting state. Standard supervised learning algorithms can be applied to learn this model, which can then be combined with $TD(\lambda)$ to learn an optimal policy.

The second approach is a model-free algorithm called Q-learning, developed by Watkins (Watkins, 1989; Watkins & Dayan, 1992), which is the subject of the next section.

4.4 Model-free Reinforcement Learning (Q-learning)

Q-learning is an online approximation of value iteration. The key to Q-learning is to replace the value function $f(s)$ with an *action-value function*, $Q(s,a)$. The quantity $Q(s,a)$ gives the expected cumulative discounted reward of performing action a in state s and then pursuing the current policy thereafter. Hence, the value of a state is the maximum of the Q values for that state:

$$f(s) = \max_a Q(s,a).$$

We can write down the Q version of the Bellman equation as follows:

$$Q(s,a) = \sum_{s'} P(s'|s,a) \left[R(s'|s,a) + \max_{a'} \gamma Q(s',a') \right].$$

The role of the Q function is illustrated in Figure 10, where it is contrasted with the value function f. In the left part of the figure, we see that the Bellman backup updates the value of $f(s)$, by considering the values $f(s')$ of states that result from different possible actions. In the right part of the figure, the analogous backup works by taking the best of the values $Q(s',a')$ and backing them up to compute an updated value for $Q(s,a)$.

The Q function can be learned by an algorithm that exploits the same insight as $TD(0)$—online sampling of the transition probabilities—and an additional idea—online sampling of the available actions. Specifically, suppose we have visited state s, performed action a, and observed the resulting state s' and immediate reward $R(s,a,s')$. We can update the Q function as follows:

$$Q(s,a) := (1-\alpha)Q(s,a) + \alpha \cdot \left(R(s,a,s') + \gamma \max_a \; Q(s',a) \right).$$

Suppose that every time we visit state s, we choose the action a uniformly at random. Then the effect will be to approximate a full Bellman backup (see Equation 4). Each value $Q(s,a)$ will be the expected cumulative discounted reward of executing action a in s, and the maximum of these values will be $f(s)$. The random choice of a ensures that

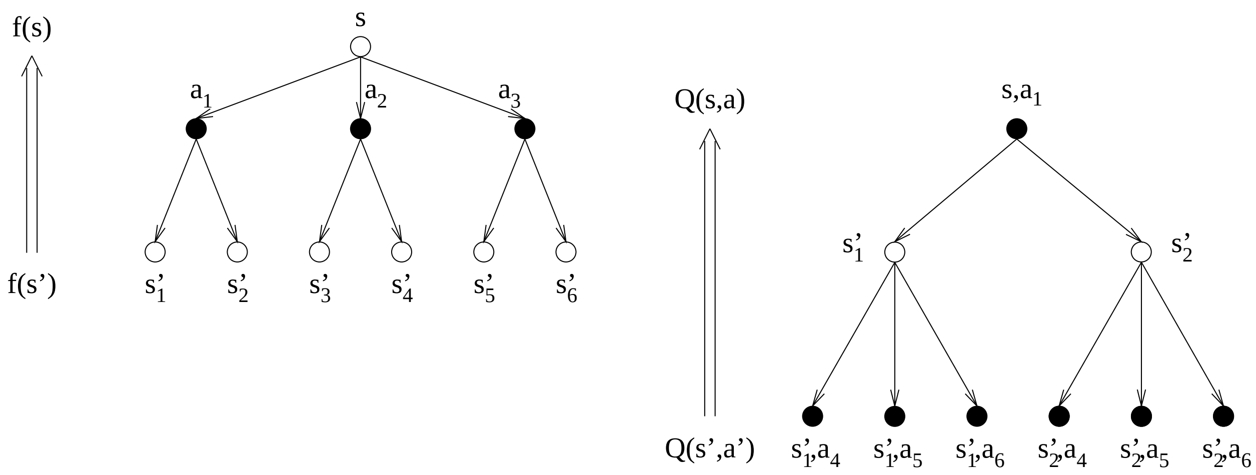

Figure 10: A comparison of the value function f and the Q function. Black nodes represent situations where the agent has chosen an action. White nodes are states where the agent has not yet chosen an action.

we learn Q values for every action available in state s. The online updates ensure that we experience the resulting states s' in proportion to their probabilities $P(s'|s,a)$.

In general, we can choose which actions to perform in any way we like, as long as there is a non-zero probability of performing every action in every state. Hence, a reasonable strategy is to choose the best action (i.e., the one whose estimated value $Q(s,a)$ is largest) most of the time but to choose a random action with some small probability. Watkins proves that as long as every action is performed in every state infinitely many times, the Q function will converge to the optimal function Q^* with probability 1.

Once we have learned Q^*, we must convert it into a policy. While this presented an insurmountable difficulty for $TD(0)$, it is trivial for the Q representation. The optimal policy can be computed as

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

Crites and Barto (1995) applied Q-learning to a problem of controlling four elevators in a 10-story office building during peak "down traffic." People press elevator call buttons on the various floors of the building to call the elevator to that floor. Once inside the elevator, they may also press destination buttons to request that the elevator stop at various floors. The elevator control decisions are (a) to decide, after stopping at a floor, which direction to go next, and (b) to decide, when approaching a floor, whether to stop at that floor or skip it. Crites and Barto applied rules to make the first decision, so the reinforcement learning problem is to learn whether to stop or skip floors. The goal of the controller is to minimize the square of the time that passengers must wait for the elevator to arrive after pressing the call button.

Crites and Barto used a team of four Q-learners, one for each of the four elevator cars. Each Q-function was represented as a neural network with 47 input features, 20 sigmoidal hidden units, and 2 linear output units (to represent $Q(s, \mathsf{stop})$ and $Q(s, \mathsf{skip})$). The immediate reward was the (negative of the) squared wait time since the previous action. They employed a form of random exploration in which exploratory actions are more likely to be chosen if they have higher estimated Q values.

Figure 11 compares the performance of the learned policy to that of eight heuristic algorithms, including the best non-proprietary algorithms. The left-hand graph shows the squared wait time, while the right-hand graph shows the percentage of passengers that had to wait more than 60 seconds for the elevator. The learned Q policy performs better than all of the other methods.

Another interesting application of Q learning is to the problem of assigning radio channels for cellular telephone traffic. Singh and Bertsekas (1997) showed that Q learning could find a much better policy than some quite sophisticated and complex published methods.

4.5 Open Problems in Reinforcement Learning

Many important problems remain unsolved in reinforcement learning, which reflects the relative youth of the field. I discuss a few of these problems here.

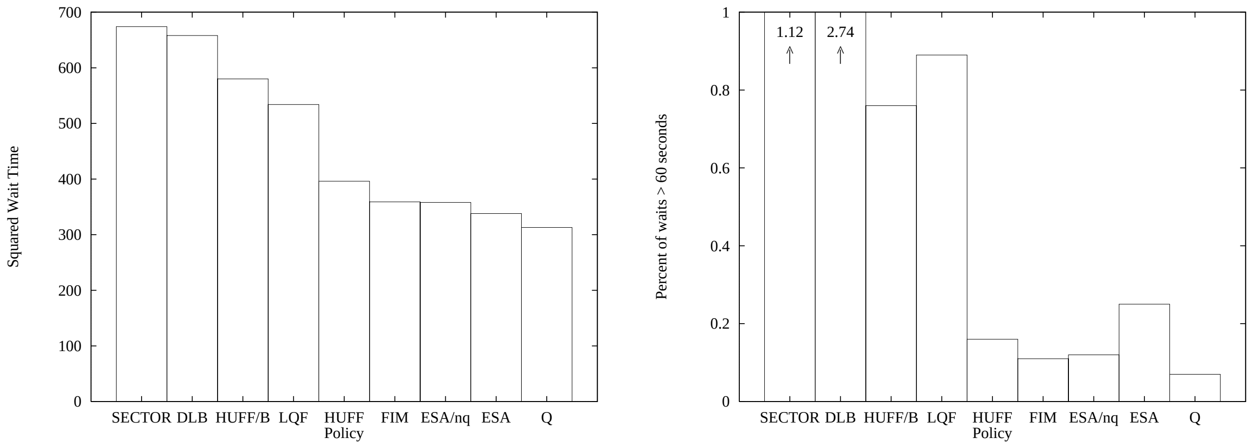

Figure 11: Comparison of learned elevator policy Q with eight published heuristic policies.

First, the use of multilayer sigmoidal neural networks for value function approximation has worked, but there is no reason to believe that such networks are well-suited to reinforcement learning. First, they tend to forget episodes (both good and bad), unless they are retrained on those episodes frequently. Second, the need to make small gradient descent steps makes learning very slow, particularly in the early stages. An important open problem is to clarify what properties an ideal value function approximator would possess and to develop function approximators with those properties. Initial research suggests that value function approximators should be "local averagers" that compute the value of a new state by interpolating among the values of previously visited states (Gordon, 1995).

A second key problem is to develop reinforcement methods for hierarchical problem solving. For very large search spaces, where the distance to the goal and the branching factor are big, no search method can work well. Often such large search spaces have a hierarchical (or approximately hierarchical) structure that can be exploited to reduce the cost of search. There have been several studies of ideas for hierarchical reinforcement learning (e.g., Singh, 1992; Dayan & Hinton, 1993; Kaelbling, 1993).

The third key problem is to develop intelligent exploration methods. Weak exploration methods that rely on random or biased random choice of actions cannot be expected to scale well to large, complex spaces. A property of the successful applications shown above (particularly backgammon and job-shop scheduling) is that even random search will reach a goal state and receive a reward. In domains where success is contingent on a long sequence of successful choices, random search has a very low probability of receiving any reward. More intelligent search methods, such as means-ends analysis, need to be integrated into reinforcement learning systems as they have been integrated into other learning architectures such as Soar (Laird, Newell, & Rosenbloom, 1987) and Prodigy (Minton, Carbonell, Knoblock, Kuokka, Etzioni, & Gil, 1989).

A fourth problem is that optimizing cumulative discounted reward is not always appropriate. In problems where the system needs to operate continuously, a better goal is to maximize the average reward per unit time. However, algorithms for this criterion are more complex and not as well-behaved. Several new methods have been put forward recently (Schwartz, 1993; Mahadevan, 1996; Ok & Tadepalli, 1996).

The fifth, and perhaps most difficult, problem is that existing reinforcement learning algorithms assume that the entire state of the environment is visible at each time step. This is not true in many applications, such as robot navigation or factory control, where the available sensors provide only partial information about the environment. A few algorithms for the solution of hidden-state reinforcement learning problems have been developed (Cassandra, Kaelbling, & Littman, 1994; Littman, Cassandra, & Kaelbling, 1995; McCallum, 1995; Parr & Russell, 1995). Exact solution appears to be very difficult. The challenge is to find approximate methods that scale well to large hidden-state applications.

Despite these very substantial open problems, reinforcement learning methods are already being applied to a wide range of industrial problems where traditional dynamic programming methods are infeasible. Researchers in the area are optimistic that reinforcement learning algorithms can solve many problems that have resisted solution by machine learning methods in the past. Indeed, the general problem of choosing actions to optimize expected utility is exactly the problem faced by general intelligent agents. Reinforcement learning provides one approach to attacking these problems.

5 Learning Stochastic Models

The final topic that I will discuss is the area of learning stochastic models. Traditionally, researchers in machine learning have sought very general-purpose learning algorithms—such as decision tree, rule, neural network, and nearest neighbor algorithms—that could efficiently search a large and flexible space of classifiers for a good fit to training data. While these algorithms are very general, they have a major drawback. In a practical problem where there is extensive prior knowledge, it can be quite difficult to incorporate this prior knowledge into these very general algorithms. A secondary problem is that the classifiers constructed by these general learning algorithms are often difficult to interpret—their internal structure may not have any correspondence to the real-world process that is generating the training data.

Over the past five years or so there has been tremendous interest in a more knowledge-based approach based on stochastic modeling. A *stochastic model* is a model that describes the real-world process by which the observed data are generated. Sometimes the terms *generative stochastic model* and *causal model* are used to emphasize this perspective. The stochastic model is typically represented as a probabilistic network—a graph structure that captures the probabilistic dependencies (and independencies) among a set of random variables. Each node in the graph has an associated probability distribution, and from these individual distributions, the joint distribution of the observed data can be computed. To solve a learning problem, the programmer designs the structure of the graph and chooses the forms of the probability distributions. This yields a stochastic model with many free parameters (i.e., the parameters of the node probability distributions). Given a training sample, learning algorithms can be applied to determine the values of the free parameters and thereby fit the model to the data. Once a stochastic model has been learned, probabilistic inference can be carried out to support tasks such as classification, diagnosis, and prediction.

More details on probabilistic networks are given in two recent textbooks: Jensen (1996) and Castillo, Gutierrez, and Hadi (1997).

5.1 Probabilistic Networks

Figure 12 is an example of a probabilistic network that might be used to diagnose diabetes. There are six variables (with their abbreviations):

- Age: Age of patient (A),
- Preg: Number of pregnancies (N),
- Mass: Body mass (M),
- Insulin: Blood insulin level (after a glucose tolerance test) (I),
- Glucose: Blood glucose level (after a glucose tolerance test) (G), and
- Diabetes: True if the patient has diabetes (D).

In a medical diagnosis setting, the first five variables would be observed and then the computer would estimate the probability that the patient has diabetes (i.e., estimate the probability that the Diabetes variable is true).

This network corresponds to the following decomposition of the joint probability distribution among the six variables:

$$P(A,N,M,I,G,D) = P(A) \cdot P(N) \cdot P(M|A,N) \cdot P(D|M,A,N) \cdot P(I|D) \cdot P(G|I,D).$$

Each node in the network corresponds to a probability distribution of the form $P(\mathsf{Node}|\mathsf{Parents})$, where the Parents of Node are the nodes with arcs pointing to Node. In other words, if we believe the network is a correct representation of the relationships among the variables, then it should be possible to factor the joint probability distribution into the product of these smaller distributions.

The structure of the network—particularly the arcs that are *absent*—can be viewed as specifying conditional independencies. Two variables A and B are conditionally independent given C if

$$P(A,B|C) = P(A|C) \cdot P(B|C).$$

In the graph, Age affects Insulin only through the Diabetes node, so Age and Insulin are conditionally independent given Diabetes. More generally, given the values of its parents, a node is independent of all other nodes in the graph except its descendents. Formally,

$$P(A,B|Parents(A)) = P(B|Parents(A)) \cdot P(A|Parents(A)),$$