1 Reinforcement Learning and its Relationship to Supervised Learning

ANDREW G. BARTO and THOMAS G. DIETTERICH
University of Massachusetts
Amherst, MA and
Oregon State University
Corvallis, OR

1.1 INTRODUCTION

The modern study of approximate dynamic programming (DP) combines ideas from several research traditions. Among these is the field of Artificial Intelligence, whose earliest period focussed on creating artificial learning systems. Today, Machine Learning is an active branch of Artificial Intelligence (although it includes researchers from many other disciplines as well) devoted to continuing the development of artificial learning systems. Some of the problems studied in Machine Learning concern stochastic sequential decision processes, and some approaches to solving them are based on DP. These problems and algorithms fall under the general heading of *reinforcement learning*. In this chapter, we discuss stochastic sequential decision processes from the perspective of Machine Learning, focussing on reinforcement learning and its relationship to the more commmonly studied supervised learning problems.

Machine Learning is the study of methods for constructing and improving software systems by analyzing examples of their behavior rather than by directly programming them. Machine Learning methods are appropriate in application settings where people are unable to provide precise specifications for desired program behavior, but where examples of desired behavior are available, or where it is possible to assign a measure of goodness to examples of behavior. Such situations include optical character recognition, handwriting recognition, speech recognition, automated steering of automobiles, and robot control and navigation. A key property of tasks in which examples of desired behavior are available is that people can perform them quite easily, but people cannot articulate exactly *how* they perform them. Hence, people can provide input-output examples, but they cannot provide precise specifications or algorithms. Other tasks have the property that people do *not* know how to perform them (or a few people can perform them only with great difficulty), but people are able to evaluate attempts to perform them, that is, to score behavior according to some performance criterion. Situations like this include playing master-level chess and controlling the nation-wide power grid in an efficient and fail-safe manner.

Machine Learning methods are also appropriate for situations where the task is changing with time or across different users, so that a programmer cannot anticipate exactly how the program should behave. For example, Machine Learning methods have been applied to assess credit-card risk, to filter news articles, to refine information retrieval queries, and to predict user browsing behavior in computer-based information systems such as the world-wide web.

Another area of application for Machine Learning algorithms is to the problem of finding interesting patterns in databases, sometimes called *data mining*. Many corporations gather information about the purchases of customers, the claims filed by medical providers, the insurance claims filed by drivers, the maintenance records of aircraft, and so forth. Machine Learning algorithms (and also, many traditional methods from statistics) can find important patterns in these data that can be applied to improve marketing, detect fraud, and predict future problems.

We begin by describing tasks in which examples of desired behavior are available.

1.2 SUPERVISED LEARNING

In *supervised learning*, the learner is given *training examples* of the form (x_i, y_i), where each input value x_i is usually an n-dimensional vector and each output value y_i is a scalar (either a discrete-valued quantity or a real-valued quantity). It is assumed that the input values are drawn from some fixed probability distribution $D(x)$ and then the output values y_i are assigned to them. The output values may be assigned by a fixed function f, so that $y_i = f(x_i)$, or they may be assigned stochastically by first computing $f(x_i)$ and then probabilistically perturbing this value with some random noise. This later, stochastic view is appropriate when the output values are assigned by a noisy process (e.g., a human judge who makes occasional errors). In either case, the goal is to correctly predict the output values of new data points x *drawn from the same distribution* $D(x)$.

For example, in optical character recognition, each input value x_i might be a 256-bit vector giving the pixel values of an 8×8 input image, and each output value y_i might be one of the 95 printable ascii characters. When, as in this case, the output values are discrete, f is called a *classifier* and the discrete output values are called *classes*.

Alternatively, in credit card risk assessment, each input might be a vector of properties describing the age, income, and credit history of an applicant, and the output might be a real value predicting the expected profit (or loss) of giving a credit card to the applicant. In cases where the output is continuous, f is called a *predictor*.

A supervised learning algorithm takes a set of training examples as input and produces a classifier or predictor as output. The set of training examples provides two kinds of information. First, it tells the learning algorithm the observed output values y_i for various input values x_i. Second, it gives some information about the probability distribution $D(x)$. For example, in optical character recognition for ascii characters, the training data provides information about the distribution of images of ascii characters. Non-ascii characters, such as greek or hebrew letters, would not appear in the training data.

The best possible classifier/predictor for data point x would be the true function $f(x)$ that was used to assign the output value y to x. However, the learning algorithm only produces a "hypothesis" $h(x)$. The difference between y and $h(x)$ is measured by a *loss function*, $L(y, h(x))$. For discrete classification, the loss function is usually the 0/1 loss: $L(y, h(x))$ is 0 if $y = h(x)$ and 1 otherwise. For continuous prediction, the loss function is usually the squared error: $L(y, h(x)) = (y - h(x))^2$. The goal of supervised learning is to choose the hypothesis h that minimizes the expected loss: $\sum_x D(x)L(y, h(x))$. Hence, data points x that have high probability are more important for supervised learning than data points that have low or zero probability.

A good way to estimate the expected loss of a hypothesis h is to compute the average loss on the training data set: $1/N \sum_{i=1}^{N} L(y_i, h(x_i))$, where N is the number of training examples. Supervised learning algorithms typically work by considering a space of hypotheses, $\mathcal{H}$, that is chosen by the designer in the hopes that it contains a good approximation to the unknown function f. The algorithms search $\mathcal{H}$ (implicitly or explicitly) for the hypothesis h that minimizes the average loss on the training set.

However, if the training examples contain noise or if the training set is unrepresentative (particularly if it is small), then an h with zero expected loss on the training examples may still perform poorly on new examples. This is called the problem of *overfitting*, and it arises when h becomes overly complex and ad hoc as the learning algorithm tries to achieve perfect performance on the training set. To avoid overfitting, learning algorithms must seek a tradeoff between the simplicity of h (simpler hypotheses are less likely to be ad hoc) and accuracy on the training examples. A standard approach is to define a *complexity measure* for each hypothesis h and to search for the h that minimizes the sum of this complexity measure and the expected loss measured on the training data.

Learning algorithms have been developed for many function classes $\mathcal{H}$ including linear threshold functions (linear discriminant analysis, the naïve Bayes algorithm, the LMS algorithm, and the Winnow algorithm; see Duda, et al., 2001), decision trees (the CART and C4.5 algorithms; see Mitchell, 1997; Breiman, et al., 1984; Quinlan, 1993), feed-forward neural networks (the backpropagation algorithm; see Bishop, 1996), and various families of stochastic models (the EM algorithm; see McLachlan & Krishnan, 1997).

Theoretical analysis of supervised learning problems and learning algorithms is conducted by researchers in the area of computational learning theory (see Kearns & Vazirani, 1994). One of the primary goals of research in this area is to characterize which function classes $\mathcal{H}$ have polynomial-time learning algorithms. Among the key results is a theorem showing that the number of training examples required to accurately learn a function f in a function class $\mathcal{H}$ grows linearly in a parameter known as the Vapnik-Chervonenkis dimension (VC-dimension) of $\mathcal{H}$. The VC-dimension of most commonly-used function classes has been computed. Another key result is a set of proofs showing that certain function classes (including small boolean formulas and deterministic finite-state automata) cannot be learned in polynomial time by any algorithm. These results are based on showing that algorithms that could learn such function classes could also break several well-known crytographic schemes which are believed to be unbreakable in polynomial time.

1.3 REINFORCEMENT LEARNING

Reinforcement learning comes into play when examples of desired behavior are not available but where it is possible to score examples of behavior according to some performance criterion. Consider a simple scenario. Mobile phone users sometimes resort to the following procedure to obtain good reception in a new locale where coverage is poor. We move around with the phone while monitoring its signal strength indicator or by repeating "Do you hear me now?" and carefully listening to the reply. We keep doing this until we either find a place with an adequate signal or until we find the best place we can under the circumstances, at which point we either try to complete the call or give up. Here, the information we receive is not directly telling us where we should go to obtain good reception. Nor is each reading telling us in which direction we should move next. Each reading simply allows us to evaluate the goodness of our current situation. We have to move around—explore—in order to decide where we should go. We are not given examples of correct behavior.

We can formalize this simple reinforcement learning problem as one of optimizing an unknown reward function R. Given a location x in the world, $R(x)$ is the reward (e.g., phone signal strength) that can be obtained at that location. The goal of reinforcement learning is to determine the location x^* that gives the maximum reward $R(x^*)$. A reinforcement learning system is not given R, nor is it given any training examples. Instead, it has the ability to take actions (i.e., choose values of x) and observe the resulting reward $R(x)$. The reward may be deterministic or stochastic.

We can see that there are two key differences from supervised learning. First, there is no fixed distribution $D(x)$ from which the data points x are drawn. Instead, the learner is in charge of choosing values of x. Second, the goal is not to predict the output values y for data points x, but instead to find a single value x^* that gives maximum reward. Hence, instead of minimizing expected loss over the entire space of x values (weighted according to $D(x)$), the goal is to maximize the reward at a single location x^*. If the reward $R(x)$ is stochastic, the goal is to maximize expected reward, but the expectation is taken with respect to the randomness in R at the single point x^*, and not with respect to some probability distribution $D(x)$.

We can formalize this simple form of reinforcement learning in terms of minimizing a loss function. The loss at a point x is the *regret* we have for choosing x instead of x^*: $L(x) = R(x^*) - R(x)$. This is the difference between the reward we could have received at x^* and the reward we actually received. However, this formulation is rarely used, because there is no way for the learner to measure the loss without knowing x^*—and once x^* is known, there is no need to measure the loss!

Given that this simple form of reinforcement learning can be viewed as optimizing a function, where does *learning* come in? The answer is long-term memory. Continuing the mobile phone example, after finding a place of good reception suppose we want to make another call from the same general area. *We go directly back to that same place, completely bypassing the exploratory search* (or we may not bother at all in the case when we were unsuccessful earlier). In fact, over time, we can build up a library of suitable spots in frequently visited locales where we go *first* when we want to make calls, and from which we possibly continue exploring. In a supervised version of this task, on the other hand, we would be directly told where the reception is best for a set of example locales. Reinforcement learning combines *search* and *long-term memory*. Search results are stored in such a way that search effort decreases—and possibly disappears—with continued experience.

Reinforcement learning has been elaborated in so many different ways that this core freature of combining search with long-term memory is sometimes obscured. This is especially true with respect to extensions of reinforcement learning that apply to sequential decision problems. In what follows, we first discuss in more detail several aspects of reinforcement learning and then specialize our comments to its application to sequential decision problems.

1.3.1 Why Call It Reinforcement Learning?

The term reinforcement comes from studies of animal learning in experimental psychology, where it refers to the occurrence of an event, in the proper relation to a response, that tends to increase the probability that the response will occur again in the same situation. The simplest reinforcement learning algorithms make use of the commonsense idea that if an action is followed by a satisfactory state of affairs, or an improvement in the state of affairs, then the tendency to produce that action is strengthened, i.e., reinforced. This is the principle articulated by Thorndike in his famous “Law of Effect” (Thorndike, 1911). Instead of the term reinforcement learning, however, psychologists use the terms *instrumental conditioning*, or *operant conditioning*, to refer to experimental situations in which what an animal actually does is a critical factor in determining the occurrence of subsequent events. These situations are said to include *response contingencies*, in contrast to Pavlovian, or classical, conditioning situations in which the animal’s responses do not influence subsequent events, at least not those controlled by the experimenter. There are very many accounts of instrumental and classical conditioning in the literature, and the details of animal behavior in these experiments are surprisingly complex. See, for example, Hergenhahn & Olson, 2001. The basic principles of learning via reinforcement have had an influence on engineering for many decades (e.g., Mendel & McClaren, 1970) and on Artificial Intelligence since its very earliest days (Minsky, 1954, 1961; Samuel 1959; Turing, 1950). It was in these early studies of artificial learning systems that the term reinforcement learning seems to have originated. Sutton and Barto (1998) provide an account of the history of reinforcement learning in Artificial Intelligence.

But the connection between reinforcement learning as developed in engineering and Artificial Intelligence and the actual details of animal learning behavior is far from straightforward. In prefacing an account of research attempting to capture more of the details of animal behavior in a computational model, Dayan (2002) stated that "Reinforcement learning bears a tortuous relationship with historical and contemporary ideas in classical and instrumental conditioning." This is certainly true, as those interested in constructing artificial learning systems are motivated more by computational possibilies than by a desire to emulate the details of animal learning. This is evident in the view of reinforcement learning as a combination of search and long-term memory discussed above, which is a an abstract computational view that does not attempt to do justice to all the subleties of real animal learning.

For our mobile phone example, the principle of learning by reinforcement is involved in several different ways depending on what grain size of behavior we consider. We could think of a move in a particular direction as a unit of behavior, being reinforced when reception improved, in which case we would tend to continue to move in the same direction. Another view, one that includes long-term memory, is that the tendency to make a call from a particular place is reinforced when a call from that place is successful, thus leading us to increase the probability that we will make a call from that place in the future. Here we see the reinforcement process manifested as the storing in long-term memory of the results of a successful search. Note that the principle of learning via reinforcement does not imply that only *gradual* or *incremental* changes in behavior are produced. It is possible for complete learning to occur on a single trial, although gradual changes in behavior make more sense when the contingencies are stochastic.

1.3.2 Behavioral Variety

Because it does not directly receive training examples or directional information, a reinforcement learning system has to actively try alternatives, process the resulting evaluations, and use some kind of selection mechanism to guide behavior toward the better alternatives. Many different terms have been used to describe this basic kind of process, which of course is also at the base of evolutionary processes: selectional (as opposed to instructional, which refers to processes like supervised learning), generate-and-test, variation and selection, blind variation and selection, and trial-and-error. These last two terms deserve discussion since there is some confusion about them. To many, blind variation and trial-and-error connote totally random, i.e., uniformly distributed, behavior patterns. But this is not what those who have used these terms have meant (e.g., Dawkins, 1985). *Blind* variation refers to the need to sometimes take actions whose consequences cannot be foreseen, that is, which represent true leaps beyond the current knowledge base. This does not mean that these actions must be randomly chosen. They can be based on a large amount of accumulated knowledge, but they cannot have consequences that can be accurately deduced from the current knowledge base. Note that even a deterministic action selection process can satisfy this requirement. Similarly, trial-and-error learning has the same meaning. Trials do not have to be random.

As a result of the need for behavioral variety, reinforcement learning involves a conflict between *exploitation* and *exploration*. In deciding which action to take, the agent has to balance two conflicting objectives: it has to exploit what it has already learned in order to perform at a high level, and it has to behave in new ways—explore—to learn more. Because these needs ordinarily conflict, reinforcement learning systems have to somehow balance them. In control engineering, this is known as the conflict between control and identification, or the problem of dual control (Feldbaum, 1965). This conflict is not present in supervised learning tasks unless the learner can influence which training examples it processes, a setting that is known as *active learning*.

At the root of this conflict is that in a reinforcement learning task a search must be conducted for something that cannot be recognized based on its intrinsic properties. For example, the property of being the better of two alternatives depends on both alternatives—it is a relative property—and a search algorithm has to examine both alternatives to decide which is the better. It is logically necessary to examine the inferior alternative. In contrast, the objectives of other types of searches depend on intrinsic properties of the members of the search space. For example, in searching for the name "Adam Smith" in a telephone directory, one can recognize the target name when one sees it because being, or not being, the target name is an intrinsic property of individual names. When a name is found that satisfies the solution property, the search stops. In a search task involving intrinsic properties, such as a supervised learning task, it is conceivable that the search can be declared accomplished after examining a single element in the search space. If the search is based on relative properties, however, this is never possible. Note that since we are speaking about the logical properties of search processes, the distinction between searching defined by relative and intrinsic properties is somewhat different than the distinction between *satisficing* and *optimizing* (Simon, 1947), which refers to more practical issues in creating a stopping criterion.

1.3.3 Converting Reinforcement Learning to Supervised Learning

Having discussed key differences between reinforcement learning and supervised learning, the question arises as to whether these differences are fundamental or merely superficial differences that can be eliminated with suitable problem reformulation. In other wards, are there ways of reducing one type of problem to the other? The first thing to note is that it is possible to convert any supervised learning task into a reinforcement learning task: the loss function of the supervised task can be used as to define a reward function, with smaller losses mapping to larger rewards. (Although it is not clear why one would want to do this because it converts the supervised problem into a more difficult reinforcement learning problem.) But is it possible to do this the other way around: to convert a reinforcement learning task into a supervised learning task?

In general, there is no way to do this. The key difficulty is that whereas in supervised learning, the goal is to reconstruct the unknown function f that assigns output values y to data points x, in reinforcement learning, the goal is to find the input x^* that gives the maximum reward $R(x^*)$.

Nonetheless, is there a way that we could apply ideas from supervised learning to perform reinforcement learning? Suppose, for example, that we are given a set of training examples of the form $(x_i, R(x_i))$, where the x_i are points and the $R(x_i)$ are the corresponding observed rewards. In supervised learning, we would attempt to find a function h that approximates R well. If h were a perfect approximation of R, then we could find x^* by applying standard optimization algorithms to h. But notice that h could be a very poor approximation to R and still be very helpful for finding x^*. Indeed, h could be any function such that the maximum value of h is obtained at x^*. This means, for example, that we do not want to use the expected loss at each training point x_i as the goal of learning. Instead, we seek a function h whose maxima are good approximations of R's maxima. There are a variety of ways of formulating this problem as an optimization problem that can be solved. For example, we can require h be a good approximation of R but that it also satisfy the following constraint: for any two training examples $(x_1, R(x_1))$ and $(x_2, R(x_2))$, if $R(x_1) > R(x_2)$ then $h(x_1)$ must be greater than $h(x_2)$. Techniques of this kind are an area of active research (Dietterich & Wang, 2003). However, note that these optimization problems are not equivalent to supervised learning problems.

1.4 SEQUENTIAL DECISION TASKS

The reinforcement learning tasks most relevant to approximate DP are sequential decision tasks. It is not an exaggeration to say that the application of reinforcement learning algorithms to these tasks accounts for nearly all of the current interest in reinforcement learning by Machine Learning researchers. In these problems, a computer program must make a sequence of decisions, where each decision changes the state of the program's environment and is followed by a numerical reward. The performance function is a measure of the total amount of reward received over a (possilby infinite) sequence of decisions. The case most commonly studied has the property that each immediate reward is zero until the end of the sequence, when it evaluates a final outcome. Imagine a computer playing the game of chess. The computer makes a long sequence of moves before it finds out whether it wins or loses the game. Similarly, in robot navigation, the robot must choose a sequence of actions in order to get from a starting location to some desired goal. We could train computers to play chess or control robots by telling them which move to make at each step. But this is difficult, tedious, and time-consuming. Furthermore, we may not know enough about the task to be able to give correct training information. It would be much nicer if computers could learn these tasks from only the final outcome—the win or loss in chess, the success or failure in robot navigation. Reinforcement learning algorithms are designed for this kind of *learning from delayed reward*.

Reinforcement learning researchers have widely adopted the framework of *Markov decision processes* (MDPs) to study sequential reinforcement learning. MDPs are discrete-time stochastic optimal control problems with a well-developed theory (see, e.g., Bertsekas, 1987). A full specification of an MDP includes the probabilistic details of how state transitions and rewards are influenced by a set of actions. The objective is to compute an *optimal policy*, i.e., a function from states to actions that maximizes the expected performance from each state, where a number of different performance measures are typically used. Given a full specification of an MDP with a finite number of states and actions, an optimal policy can be found using any of several stochastic DP algorithms, although their computational complexity makes them impractical for large-scale problems.

Reinforcement learning for sequential decision tasks consists of a collection of methods for approximating optimal policies of MDPs, usually under conditions in which a full specification of the MDP is unavailable. A typical reinforcement learning system learns a task by repeatedly performing it—that is, it makes moves in chess or issues control commands to a robot. Before each move, the algorithm can examine the current state, s of the environment (i.e., the current board position in a chess game or the current sensor inputs from the robot together with any other relevant robot-internal information) and then choose and execute an action a (i.e., a chess move or a robot command). The action causes the environment to change to a new state s'. After each state transition, the learning system receives a reward, $R(s, a, s')$. In chess, the reward is zero until the end of the game, where it is 1 (win), 0 (draw), or -1 (loss). In robot navigation, there is typically a large positive reward for reaching the goal position and a small negative reward for each step. There may also be negative rewards for using energy, taking time, or bumping into walls or other obstacles.

There are many alternative approaches to approximating optimal policies. The most direct approach is to directly learn a policy. In this approach, a space of possible policies is defined, usually be defining a parameterized family of policy functions that are continuously differentiable with respect to the parameters. Given a particular policy (corresponding to a particular parameter setting), there are algorithms that can estimate the gradient of the expected performance (i.e., the expected total reward, the expected discounted total reward, or the expected reward per step) of the policy with respect to its parameters by performing online trial executions of the policy (e.g., Baxter & Bartlett, 2001; Baxter et al., 2001).

A somewhat less direct approach is to learn a *value function*, V, which assigns a real number $V(s)$ to each state s indicating how valuable it is for the system to be in that state. A closely-related method learns an *action-value function*, Q, where $Q(s, a)$ tells how valuable it is to do action a in state s. In either case, the value is an estimate of the total amount of reward that will accumulate over the future starting in the specfied state. (The counterpart of a value function in a cost-minimizing formulation is sometimes called the “cost-to-go” function.) Once the system has learned a good approximation of the value function, it can execute an improved policy by choosing actions that have higher values or lead to states that have higher values. For example, given the action-value function Q, the policy for state s can be computed as the action a that maximizes $Q(s, a)$. As this process continues, one expects the policy to improve toward optimality in an approximation of the policy improvement DP procedure. Reinforcement learning algorithms that learn value functions often update their policies before the value functions of their current policies have been fully learned. This allows them to decide on actions quickly enough to meet time constraints.

The most indirect approach to approximating an optimal policy is to learn a model of the MDP itself through a system identification procedure. For DPs this involves learning the reward function $R(s, a, s')$ and the transition probability function $P(s'|s, a)$ (i.e., the probability that the environment will move to state s' when action a is executed in state s). These two functions can be learned by interacting with the environment and using a system identification procedure. Each time the learner observes state s, performs action a, receives reward r and moves to the resulting state s', it obtains training examples for $P(s'|s, a)$ and $R(s, a, s')$. These examples can be given to supervised learning algorithms to learn P and R. Once these two functions have been learned with sufficient accuracy, DP algorithms can be applied to compute the optimal value function and optimal policy. Methods that learn a model are known as "model-based" methods. They typically require the fewest number of exploratory interactions with the environment, but they also typically do not scale well to very large problems.

These three approaches are not mutually exclusive. There are policy-search methods that learn partial models to help compute gradients (Wang & Dietterich, 2003) and value function methods that learn partial models and perform incremental DP (Barto et al., 1995; Moore & Atkeson, 1993).

The algorithms that may scale best are model-free algorithms for estimating value functions, such as the Temporal Difference algorithm (Sutton, 1988, 1996). Interestingly, the temporal difference family of algorithms can be viewed as supervised learning algorithms in which the training examples consist of $(s, \hat{V}(s))$ pairs, where s is a state and $\hat{V}(s)$ is an approximation of the value of state s. They are not true supervised learning algorithms, because the $\hat{V}(s)$ values are not provided by a teacher but instead are computed from the estimated values of future states. For this reason, they are sometimes called "bootstrapping" methods. The other reason that these algorithms are not true supervised learning algorithms is that the probability distribution over the states s is not a fixed distribution $D(s)$. Instead, the distribution depends on the current value function $\hat{V}$ and the current policy for choosing exploratory actions. Despite these differences, many algorithms from supervised learning can be applied to these temporal difference algorithms (Bradtke & Barto, 1996; Dietterich & Wang, 2002; Lagoudakis & Parr, 2003).

Reinforcement learning methods that use value functions are closely related to DP algorithms, which successively approximate optimal value functions and optimal policies for both deterministic and stochastic problems. Details are readily available elsewhere (see, e.g., Sutton & Barto, 1998). Most reinforcement learning algorithms that estimate value functions share a few key features:

1. Because conventional DP algorithms require multiple exhaustive "sweeps" of the environment state set (or a discretized approximation of it), they are

not practical for problems with very large finite state sets or high-dimensional continuous state spaces. Instead of requiring exhaustive sweeps, reinforcement learning algorithms operate on states as they occur in actual or simulated experiences in controlling the process. It is appropriate to view them as *Monte Carlo* DP algorithms.

2. Whereas conventional DP algorithms require a complete and accurate model of the process to be controlled, many reinforcement learning algorithms do not require such a model. Instead of computing the required quantities (such as state values) from a model, they estimate these quantities from experience. However, as described above, reinforcement learning methods can learn models in order to improve their efficiency.

3. Conventional DP algorithms use lookup-table storage of values for all states, which is impractical for large problems. Reinforcement learning algorithms often use more compact storage schemes in the form of parameterized function representations whose parameters are adjusted through adaptations of various function approximation methods.

1.4.1 Reinforcement Learning and other Approximate DP Methods

While there is not a sharp distinction between reinforcement learning algorithms and other methods for approximating solutions to MDPs, several features tend to be associated with reinforcement learning. The most conspicuous one is that computation in reinforcement learning takes place during interaction between an active decision maker and its environment. The computational process occurs while the decision maker is engaged in making decisions as opposed to being an off-line batch process. Artificial Intelligence researchers say that the decision maker is “situated” in its environment. This feature arises from an underlying interest in *learning*, as we see it accomplished by ourselves and other animals, and not merely in general computational methods. Often this interaction is only virtual, as a simulated learning agent interacts with a simulated environment, but even in this case, the processes of *using* and *acquiring* knowledge are not separated into two distinct phases.

As a result of this emphasis, the objective of a reinforcement learning algorithm is not necessarily to approximate an optimal policy, at least not uniformly across the state space, as in conventional approaches to MDPs. It is more accurate to think of the objective from the active agent’s point of view: it is to obtain as much reward over time as possible. Since an agent usually visits states non-uniformly while it is behaving, the approximation error is weighted by the agent’s state-visitation distribution, a so-called *on-policy distribution* (Sutton & Barto, 1998). This is possible due to the situated nature of the reinforcement learning process. In some problems it confers significant advantages over conventional DP algorithms because large portions of the state space can be largely irrelevant for situated behavior.

A second feature that tends to be associated with reinforcement learning is the lack of complete knowledge of the MDP in question. The process of computing an optimal policy, or an approximately-optimal policy, given complete knowledge of an MDP’s state transition and reward probabilities is considered to be more of a *planning* problem than a learning problem. However, there are many applications of reinforcement learning that make use of so-called *generative models*. These are simulation models of the MDP. Given a chosen state s and action a, a generative model can produce a next state s' sampled according to the probability transition function $P(s'|s,a)$ and an immediate reward generated according to $R(s,a,s')$. Some generative models permit the learner to jump around from one state to another arbitrarily, while other generative models can only simulate continuous trajectories through the state space. In principle, a generative model contains the same information as knowing P and R, but this information is not available in an explicit form and therefore cannot be used directly for DP. Instead, the generative model is applied to simulate the interaction of the learner with the environment, and reinforcement learning algorithms are applied to approximate an optimal policy. The advantage of generative models from an engineering perspective is that it is often much easier to design and implement a generative model of an application problem than it is to construct an explicit representation of P and R. In addition, learning from a generative model (e.g., of a robot aircraft) can be faster, safer, and cheaper than learning by interacting with the real MDP (e.g., a real robot). Using generative models in approximating solutions to MDPs is closely associated with reinforcement learning even though learning from on-line experience in the real world remains a goal of many reinforcement learning researchers.

1.5 SUPERVISED LEARNING FOR SEQUENTIAL DECISION TASKS

For the same reasons that non-sequential reinforcement learning cannot be reduced to supervised learning, sequential reinforcement learning can also not be reduced to supervised learning. First, the information provided by the environment (the next state and the reward) does not specify the correct action to perform. Second, the goal is to perform the optimal action in those states that are visited by the optimal policy. But this distribution of states is not a fixed distribution $D(s)$, but instead depends on the actions chosen by the learner. However, there is an additional reason why reinforcement learning in sequential decision tasks is different from supervised learning: In sequential decision tasks, *the agent must suffer the consequences of its own mistakes.*

Consider the problem of learning to steer a car down a highway. We can view this as a supervised learning problem in which an expert human teacher drives the car down the highway and the learner is given training examples of the form (s,a), where s is the current position of the car on the road (e.g., its position relative to the edges of the lane) and a is the steering action chosen by the teacher. We could now view this as a supervised learning problem where the goal is to learn the function $a = f(s)$. This approach has been termed "behavioral cloning" (Sammut??), because we wish to "clone" the behavior of a human expert.

Note that the distribution of states $D(s)$ contains only those states (i.e., those positions of the car) that are visited by a good human driver. If the function f can be learned completely and correctly and the car is started in a good state s, then there is no problem. But supervised learning is never perfect. Let h be the hypothesis output by the supervised learner, and let s_1 be a state where $h(s_1) \neq f(s_1)$. In this state, the learner will make a mistake, and the car will enter a new state s_2 chosen according to $P(s_2|s_1, h(s_1))$. For example, s_1 might be a state where the right wheels of the car are on the edge of the highway, and state s_2 might be a state where the right wheels are off the road. Now this state was never observed during training (because the human teacher would never make this mistake, so $D(s_2) = 0$). Consequently, the learner does not know how to act, and the car could easily leave the road completely and crash. The point is that even if h is 99.99% correct on the distribution $D(s)$, even a single state s_1 where h is wrong could lead to arbitrarily bad outcomes. Reinforcement learning cannot be solved by supervised learning, even with a perfect teacher. It is interesting to note that the ALVINN project (Pomerleau, 1991) attempted to address exactly this problem by applying domain knowledge to generate synthetic training examples for a wide range of states including states where the car was far off the road.

Because reinforcement learning occurs online through interacting with the environment, the learner is forced to learn from its own mistakes. If the car leaves the road and crashes, the learner receives a large negative reward, and it learns to avoid those actions. Indeed, a reinforcement learning system can learn to avoid states that could potentially lead to dangerous states. Hence, the learner can learn to avoid states where the wheels get close to the edges of the lane, because those states are "risky." In this way, a reinforcement learning system can learn a policy that is better than the best human expert. This was observed in the TD-gammon system, where human experts changed the way they play certain backgammon positions after studying the policy learned by reinforcement learning (Tesauro, 1992, 1994, 1995).

1.6 CONCLUDING REMARKS

In this article, we have attempted to define supervised learning and reinforcement learning and clarify the relationship between these two learning problems. We have stressed the differences between these two problems, because this has often been a source of confusion. Nonetheless, there are many similarities. Both reinforcement learning and supervised learning are statistical processes in which a general function is learned from samples. In supervised learning, the function is a classifier or predictor; in reinforcement learning, the function is a value function or a policy.

A consequence of the statistical nature of reinforcement learning and supervised learning is that both approaches face a tradeoff between the amount of data and the complexity of the function that can be learned. If the space of functions being considered is too large, the data will be overfit, and both supervised and reinforcement learning will perform poorly. This is manifested by high error rates in supervised learning. In reinforcement learning, it is manifested by slow learning, because much more exploration is needed to gather enough data to eliminate overfitting.

Another similarity between reinforcement learning and supervised learning algorithms is that they both often make use of gradient search. However, in supervised learning, the gradient can be computed separately for each training example, whereas in reinforcement learning, the gradient depends on the relative rewards of two or more actions.

We note that the view we present of the key features distinguishing reinforcement learning from other related subjects leaves room for credible alternatives. Researchers do not thoroughly agree on these issues, and it is not clear that striving for definitive definitions serves a useful purpose. Modern problem formulations and algorithms can significantly blur some of these distinctions, or even render them irrelevant. Nevertheless, we hope that our discussion can serve as a useful guide to the core ideas behind reinforcement learning and their relationship to the fundamental ideas of supervised learning.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. ECS-0218125. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

1. A. G. Barto, A. G., Bradtke, S. J, and Singh, S. P. (1995). Learning to Act using Real-Time Dynamic Programming, *Artificial Intelligence*, 72: 81–138.

2. Baxter, J. and Bartlett, P. L. (2001). Infinite-Horizon Gradient-Based Policy Search, *Journal of Artificial Intelligence Research*, 15: 319–350.

3. Baxter, J., Bartlett, P. L., and Weaver, L. (2001). Infinite-Horizon Gradient-Based Policy Search: II. Gradient Ascent Algorithms and Experiments, *Journal of Artificial Intelligence Research,* 15: 351–381.

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

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

6. Bishop, C. M. (1996). *Neural Networks for Pattern Recognition*. Oxford University Press. Oxford, England.

7. Boyan, J. A. (1999). Least-Squares Temporal Difference Learning. In Bratko, I., and Dzeroski, S., eds., Machine Learning: Proceedings of the Sixteenth International Conference (ICML), 1999.

8. Bradtke, S. J. and Barto, A. G. (1996). Linear Least–squares Algorithms for Temporal Difference Learning, *Machine Learning*, 22: 33–57.

9. Breiman, L, Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA.

10. Dayan, P. (2003). Motivated Reinforcement Learning. In Dietterich, T. G., Becker, S., Ghahramani, Z. (Eds.) *Advances in Neural Information Processing Systems 14, Proceedings of the 2002 Conference*, pp. 11–18. MIT Press, Cambridge, MA.

11. Dietterich, T. G. and Wang, X. (2003). Batch value function approximation via support vectors. In Dietterich, T. G., Becker, S., Ghahramani, Z. (Eds.) *Advances in Neural Information Processing Systems 14, Proceedings of the 2002 Conference*, pp. 1491–1498. MIT Press, Cambridge, MA.

12. Dawkins, R. (1986). *The Blind Watchmaker*. Norton, New York.

13. Duda, R. O., Hart, P. E., and Stork, D. G. (2001). *Pattern Classification, Second Edition*. Wiley, New York.

14. Feldbaum, A. A. (1965). *Optimal Control Systems*. Academic Press, New York.

15. Hergenhahn, B. R. and Olson, M. H. (2001). *An Introduction to Theories of Learning* (Sixth Edition). ÊPrentice Hall, Upper Saddle River, NJ.

16. Kearns, M. J., Vazirani, U. V. (1994). *An introduction to computational learning theory.* MIT Press, Cambridge, MA.

17. Lagoudakis, M. G. and Parr, R. (2003). Reinforcement Learning as Classification: Leveraging Modern Classifiers. In Fawcett, T. G., Mishra, N. (Eds.) *Proceedings, Twentieth International Conference on Machine Learning*, pp. 424–431. AAAI Press, Menlo Park, CA.

18. McLachlan, G. J., Krishnan, T. (1997). *The EM Algorithms and Extensions*. John Wiley & Sons, Inc. New York.

19. Mendel, J. M. and McLaren, R. W. (1970). Reinforcement Learning Control and Pattern Recognition Systems. In: Mendel, J. M. and Fu, K. S. (eds.) *Adaptive, Learning and Pattern Recognition Systems: Theory and Applications*, pp. 287—318. Academic Press, New York.

20. Minsky, M. L. (1954). *Theory of Neural-Analog Reinforcement Systems and its Application to the Brain-Model Problem*. Ph.D. dissertation, Princeton University.

21. Minsky, M. L. (1961). Steps Toward Artificial Intelligence, *Proceedings of the Institute of Radio Engineers*, 49: 8–30, Reprinted in E. A.Feigenbaum andJ. Feldman (eds.) *Computers and Thought*, pp. 406–450. McGraw-Hill, New York, 1963.

22. Mitchell, T. (1997). *Machine Learning*. McGraw Hill.

23. Moore, A. W. and Atkeson, C. G. (1993). Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time, *Machine Learning*, 13: 103–130.

24. Pomerleau, D. A. (1991). Efficient Training of Artificial Neural Networks for Autonomous Navigation, *Neural Computation*, 3: 88–97.

25. Quinlan, J. R. (1993). *C4.5: Programs for Empirical Learning*. Morgan Kaufmann, San Francisco, CA.

26. Samuel, A. L. (1959). Some Studies in Machine Learning Using the Game of Checkers, *IBM Journal on Research and Development*, 3: 211–229. Reprinted in E. A. Feigenbaum and J. Feldman (eds.) *Computers and Thought*, pp. 71–105, McGraw-Hill, New York, 1963.

27. Simon, H. A. (1947) *Administrative Behavior*. Macmillan, New York.

28. Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

29. Sutton, R. S., (1988). Learning to Predict by the Method of Temporal Differences, *Machine Learning*, 3: 9–44.

30. Sutton, R. S., (1996). Generalization in Reinforcement Learning: Successful Examples Using Coarse Coding. In Touretzky, D. S., Moser, M. C., Hesselmo, M. E. (Eds.), *Advances in Neural Information Processing Systems, Proceedings of the 1995 Conference*, pp. 1038–1044. MIT Press, Cambridge, MA.

31. Tesauro, G. J. (1992). Practical Issues in Temporal Difference Learning, *Machine Learning*, 8: 257–277.

32. Tesauro, G. J. (1994). TD–Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play, *Neural Computation*, 6: 215–219.

33. Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, 28: 58–68.

34. Thorndike, E. L. (1911). *Animal Intelligence*. Hafner, Darien, Conn.

35. Turing, A. M. (1950). Computing Machinery and Intelligence. *Mind*, 59: 433—460. Reprinted in E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pp. 11—35, McGraw-Hill, New York, 1963.

36. Wang, X. and Dietterich, T. G. (2003). Model-Based Policy Gradient Reinforcement Learing. In Fawcett, T. G., Mishra, N. (Eds.) *Proceedings, Twentieth International Conference on Machine Learning*, pp. 776–783. AAAI Press, Menlo Park, CA.