Reinforcement Learning

Ryan P. Adams*
COS 324 – Elements of Machine Learning
Princeton University

Here we will look at several methods for reinforcement learning, and discuss two important issues: the exploration-exploitation tradeoff and the need for generalization. Finally we will look at some applications.

1 Reinforcement Learning

Recall our formulation of the agent's interaction with the environment. The agent receives a percept, which tells it something about the state of the environment. The agent then takes an action, which probabilistically causes the state of the world to change. The agent then receives a new percept, and so on. Each time the world is in a state, the agent receives a reward. We captured this situation with a Markov decision process (MDP). We considered planning algorithms, by which the agent could work out the best thing to do in each state given its model of how the world works.

Let's now turn to the case where the agent does not know the way the world works. That is, the agent is not provided with a probabilistic model of the environment. We will assume the agent knows the possible states and the possible actions. We will also assume that the agent knows the current state of the world when it makes a decision. The interaction story is as before: the world begins in some state; the agent chooses an action; the action produces a change in the state of the world, and the agent gets a reward; and so on. The only thing that is different is the agent's knowledge: the agent does not know (at least initially) the transition or reward model of the world, and so it cannot compute the optimal policy.

However, one might hope that the agent would be able to *learn* a good policy, based on rewards received and states observed. If the agent gets a reward it should learn that it must have done something good to earn the reward. It should therefore try to repeat the action when it has an opportunity to do so, in order to get the reward again. Similarly, if the agent gets a punishment (i.e., a negative reward) it should learn that it did something wrong, and try in future to avoid the action that led to the punishment. This is the basic idea of *reinforcement learning*.

You've probably heard of reinforcement learning from psychology, where it is an old and accepted fact that animals can learn from the rewards and punishments that they get. For example, from studies of rats in mazes, that show that rats can learn to navigate a maze when the only feedback

*These notes are adapted from Harvard CS181 course notes by Avi Pfeffer and David Parkes.

they get is a piece of cheese at the end of the maze. We're going to show that reinforcement learning has a computational basis, and that agents can perform quite well using reinforcement learning. In fact, there have been some remarkable successes. For example, a world-class backgammon player was constructed using reinforcement learning (RL).

There are two things that make reinforcement learning in domains with repeated interaction challenging. Suppose you are playing a game like backgammon. You get a reward or punishment at the end of the game, depending on whether you won or lost. The problem is this – even though you get your reward after the final move of the game, it is probably not that final move that really led to a win. There were probably a series of decisions that you made earlier that eventually led to you winning. The question is this: how do you determine which decisions were actually responsible for leading to the win? This is the **credit assignment** problem: how to assign credit or blame for rewards or punishments.

The second issue is that an agent in a reinforcement learning process has to determine what action to take, even while it is learning. It receives rewards and punishments even as it is learning. Imagine that you have a new job. Certainly your employers will give you some time to learn the ropes, but they will expect you to start being productive before you have learned everything there is to know about the job. In particular, a reinforcement learning agent needs to decide what to do, taking into account the effect of its action on its immediate rewards and future state, but also taking into consideration the need to learn for the future. This issue is known as th **exploration-exploitation tradeoff**.

There are two main approaches to reinforcement learning:

- The first approach is called the **model-based** approach. In the model-based approach, the agent explicitly learns a model of the MDP in which it is operating. As it learns, it can solve the current (estimated) model using a planning algorithm such as value iteration or policy iteration, while still being sure to continue exploring enough to keep learning.

- The second approach is **model-free**. Rather than learning a model, the agent tries to learn a policy directly: learn how to behave without learning a model. One popular algorithm to do this is called **Q-learning**, because the agent learns the Q function that tells it the value of taking each action in each state (and then behaving optimally). Model free approaches are very simple but they tend to be slow compared to model-based approaches.

A hybrid approach combines the two: learn a model and the value function (or Q-function) at the same time. One algorithm for implementing this is *temporal-difference value learning* (as opposed to Q-learning, which is a temporal-difference *value-action* learning method.) Another algorithm that combines the two approaches is *Dyna-Q*, in which Q-learning is augmented with extra value-update steps. An advantage of these hybrid methods over straightforward model-based methods is that solving the model can be expensive, and also if your model is not reliable it doesn't make much sense to solve it. An advantage over pure model-free approaches is that they make better use of information and allow for faster learning.

2 Model-Based Reinforcement Learning

The first approach to reinforcement learning is simple: just learn the transition and reward models. In the model based approach, *the credit assignment problem is circumvented*. Rather than using the rewards to directly learn to repeat or avoid certain actions, the rewards are used to learn what states of the world are good or bad. The agent also learns how its actions affect the state of the world. Since the process of determining an optimal policy takes into account long-term effects of actions, it will take actions that will help it achieve a reward a long way in the future.

2.1 Learning

We'll start with the reward model. First, a technical comment. In the original MDP formulation, we assumed a deterministic reward model. That is, the reward the agent gets for taking an action in a state is fixed, and always the same. This may be an unrealistic assumption in general. A more realistic model would specify that for any action in any state, there is a probability distribution over the possible reward.

Why did we assume a deterministic reward model in the MDP framework? Because an MDP with a probabilistic reward model can be converted into one with a deterministic reward model. The reason is that all we care about is maximizing the agent's expected future reward, and that depends only on the expected reward in each state, and not on the probability distribution over rewards. So we can replace a probabilistic reward model, by a deterministic model in which $R(s, a)$ is the expected immediate reward for action a in state s.

Still, when learning a reward model, we may get different rewards at different times for the same action and state, because in reality the reward received is non-deterministic. However, we only need to keep track of the expected reward.

To do this, we maintain a running average of the rewards observed upon taking each action in each state. All we need to do is maintain a count $N(s, a)$ of the number of times we have visited state s and taken action a, and $R^{\text{total}}(s, a)$, the total reward accumulated in all times action a was taken in state s. Our maximum likelihood estimate for the expected reward for action a in state s is simply $R^{\text{total}}(s, a)/N(s, a)$.

As for the transition model, that is also very simple to learn. For each state s and action a, we need to learn $P(s' \mid s, a)$, the probability of reaching state s' by taking action a in state s. To do this, we store $N(s, a)$ and also $N(s, a, s')$, the number of times a transition from s to s' happened on action a. The maximum likelihood estimate for $P(s' \mid s, a)$ is then $N(s, a, s')/N(s, a)$. Of course we are just doing maximum likelihood learning here, and more sophisticated (e.g., MAP) approaches can also be used.

2.2 Planning and Acting

Every so often, the agent will want to re-plan given its current model. This can be achieved through standard approaches such as value iteration or policy iteration. Moreover, the agent will also generally choose not to follow the optimal policy given the current model exactly, because of considerations of needing to *explore* the environment. We return to this in a later section because the issues are similar across model-based and model-free reinforcement learning.

One issue to address using the model-based approach is how often the agent should solve the MDP and update its policy. *How often should the agent re-plan?* A typical approach is to work in *epochs*, with an epoch consisting of a sequence of trials in an "episodic" problem such as driving to the airport or just a sequence of periods in a "continuous" problem such as driving around a city. An agent does not re-plan during an epoch, but replans at the end of every epoch. The question is how to tune the length of an epoch, and thus how frequently to plan. The tradeoff is between additional computation (planning is difficult) and better use of information (more frequent planning will lead to better policies more quickly.)

One extreme is to update the estimates of the transition and reward model every time it takes an action, and immediately solve the updated MDP. To make this reasonably efficient, one can simply use the previous value function as the starting point for value iteration. Since the new value function should be close to it, the algorithm should converge very quickly. Similarly, one can use the old policy as the starting point for policy iteration. Since in many cases the optimal policy does not change as a result of one update, the policy iteration algorithm will often converge in one iteration. This is quite powerful: we can even imagine replanning *every* period because of this observation. Still, model-based approaches can be infeasible because planning while learning can simply take too long in very time sensitive domains.

3 Model-Free Reinforcement Learning

The second approach to reinforcement learning dispenses with learning a model, and tries to directly learn the value of taking each action in each state. This approach is a *model-free* approach.

Recall that for the purpose of solving MDPs, we defined the function $Q(s, a)$ which provides the expected value of taking action a in state s under the optimal policy forward from the next state. The Q value incorporates both the immediate reward and expected value of the next state reached. With knowledge of a Q value, the agent does not need a model to decide how to act: it can simply keep acting according to the action with the maximum Q-value in the current state.

A popular model-free learning algorithm is the *Q-learning* algorithm. This provides a method to learn the Q function directly without needing to also learn a probabilistic model of the environment. This provides two advantages over model-based approaches: it is no longer necessary to learn a transition model of size $O(N^2M)$ for N states and M actions. Moreover, it avoids the need to perform planning while learning, which can be infeasible in complex domains. On the other hand, we will see that Q-learning can be *very slow* in terms of the number of periods of experience required to learn a good policy, and much more slow to learn a good policy than model-based RL.

3.1 Q-Learning

How does an agent learn the Q function? Expanding the definition of the Q function from the Bellman equations, we have

$$\begin{aligned} Q(s,a) &= R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) \max_{a' \in \mathcal{A}} Q(s',a') \\ &= R(s,a) + \gamma\, \mathbb{E}_{s'} \left[\max_{a' \in \mathcal{A}} Q(s',a') \right] \\ &= \mathbb{E}_{s'} \left[R(s,a) + \gamma \max_{a' \in \mathcal{A}} Q(s',a') \right] \end{aligned}$$

Our goal then, is to estimate $Q(s,a)$ as the expectation, where s' is drawn from $P(s' \mid s,a)$, of $R(s,a) + \max_{a'} Q(s',a)$. But we must do this without having access to the model of the probabilistic transition function.

This is possible because each time we actually take action a from state s we observe a transition to s' and receive a reward r. This gives us a sample from $P(s' \mid s,a)$. We can use this sample for updating our old estimate of $Q(s,a)$. Specifically, on transitioning from s to s' under action a and receiving reward r, the following update rule used in Q-learning:

$$Q(s,a) \quad \leftarrow \quad Q(s,a) + \alpha \left[(r + \gamma \max_{a' \in \mathcal{A}} Q(s',a')) - Q(s,a) \right], \tag{1}$$

where $0 < \gamma < 1$ is the discount factor and $0 < \alpha < 1$ is the *learning rate*.

To get the new estimate of $Q(s,a)$, we move the current estimate by some amount that depends on the error between $Q(s,a)$ and the target value that we find in the new state. The target value is $r + \gamma \max_{a'} Q(s',a')$, which provides a sample of $Q(s,a)$. The rule adjusts towards this, with learning rate α determining how much of an effect the new sample has on the current estimate. If α is large, we will adjust quickly but may not converge. If α is small then we will adjust slowly, but learning may converge. A natural thing to do is to decrease α gradually as the number of samples of $Q(s,a)$ increases.

3.2 Temporal Difference Learning

The general form of update that we see here, of which Q-learning provides an example, is

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize}[\text{Target} - \text{OldEstimate}]$$

This is the method of *temporal difference* (TD) learning, because the new estimate is adjusted to try to reduce the difference to the target value in the state reached in the next time period. We will see in the next section, a variation on TD learning in which the value function and not the Q-function is learned.

3.3 Planning and Acting

Every period, a Q-learner will select a new action, transition to a new state and receive a reward, and then update the Q-value in the state from where it has transitioned. But, how should an action be selected? As with model-based RL, there is a decision to make about whether to be greedy and just exploit – always selecting the action with the maximum Q value in the current state, or fold in some exploration and perhaps discover more information about the world that can be used for greater long-term rewards. Eventually, when we've learned everything there is to know about the underlying MDP, we want to just do planning and act optimally according to the best possible policy. Q-learning has the following two theoretical properties:

(i) If every state-action pair (s, a) is visited an unbounded number of times and the learning rate α is "eventually small enough" then the Q-values converge in the limit

(ii) If we exploit the Q-values in the limit, then the policy converges to the optimal policy in the the limit.

The first property (i) is a bit tricky, it says that the learning rate should allow for learning but "eventually be small enough." We provide a brief comment on this below. In practice, these two requirements (i) and (ii) are typically achieved by: 1) having a distinct learning rate for each state/action pair, and having that rate be $\alpha_k(s, a) = 1/k$ where k is the number of times action a has been taken from state s; 2) adopting a so-called "ϵ-greedy" policy in which the optimal action is taken with probability $1 - \epsilon$, but with probability ϵ, a uniformly random action is taken to induce exploration. In order to get to (ii) it is common to take $\epsilon = 1/t$, where t is the number of time periods (or perhaps number of trials in an episodic environment) that the agent has experienced.

Note that a different learning rate is assumed here for each state action pair, and that ϵ-greedy learning is adopted but with an ϵ that decays over time. Taken together, these two properties provide GLIE, and convergence to an optimal policy in the limit. Don't be fooled though, Q-learning can still be very slow to converge!

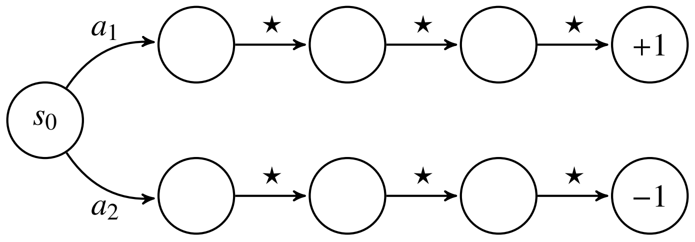
The technical properties required for α to allow for learning but be "eventually small enough" are that

$$\sum_{k=0}^{\infty} \alpha_k(s, a) = \infty \qquad \sum_{k=0}^{\infty} \alpha_k^2(s, a) < \infty,$$

which are satisfied for example by a learning rate that varies as $1/k$, but not by some fixed α. The first summation ensures there is enough learning, the second summation ensures eventually convergence.

3.4 Discussion: Credit Assignment

How does Q-learning solve the credit assignment problem? The answer is that the rewards are eventually propagated back through the Q function to all the states that lead to good reward. Consider the following situation:



Here the critical decision happens early: choosing a_1 at the first step eventually leads to a reward of $+1$, while a_2 leads to -1. This situation models a critical point in the middle of a game where you have to make a decision that eventually will lead to a win or a loss. The problem is that the information only propagates one state back in the chain every new episode. This can make learning slow – the agent is not using all the available information in the same way as was possible with model-based RL.

There is an additional concern, which is that there is a bias of Q-learning in favor of more quickly propagating positive rewards than negative rewards. For positive rewards, the number of trials required to filter backwards is linear in the number of intervening states. But for negative rewards it is exponential, because in each state every possible other action must also be ruled out before updating with a negative Q-value. This is because of the "max" in the Q-learning update rule: it is first necessary to update the Q-values to indicate that *every* action is bad before a bad Q-value is sent "upstream" (i.e., to an earlier state in the sequence.)

This asymmetry leads to the result that while positive rewards are propagated backwards reasonably quickly, punishments (i.e., negative rewards) are propagated much more slowly. A punishment will only be propagated back from a state s' to a previous state s only if it is understood to be unavoidable that the punishment will be received once s' is reached – i.e., only if it shows up in $Q(s', a)$ for every action a. For this reason, it can take many trials to discover that a punishment is unavoidable and learn that an earlier action is a bad action.

4 Hybrid Approaches

There are good arguments for both model-based and model-free approaches. The model-free approach is very simple. In addition, there is a good computational argument for it: solving MDPs is expensive. It can be done if the state space is not too huge, but shouldn't be done too often. Model-based learning, on the other hand, requires that a new policy be computed regularly as the model is updated. With Q-learning, the agent just needs to look up the Q values for the current state in order to decide what to do. On the other hand, Q-learning is not taking full advantage of the data that it is getting. Since every transition provides information about the transition model, doesn't it make sense to use that information and actually learn the model? For this reason it can be extremely slow to learn a policy with Q-learning. We explore a few simple variations that provide some intermediate approaches, and often enjoy good properties in practice.

4.1 TD-Value Learning

Q-learning, as discussed above, is a method of temporal difference value-action learning. Literally, it is learning a model for the value of different actions in each state. But what if rather than learn Q-values we learn the state-specific $V(s)$ values instead? This is temporal difference value learning.

One thing that is apparent is that we would also need to learn a model of the transition model, $P(s' \mid s, a)$, and the reward $R(s, a)$, in order to be able to know how to act. Just knowing $V^\star(s)$ for the optimal policy $\pi^\star(s)$ is not quite enough.

The TD-value learning method tries to combine the advantages of both the model-based and model-free approaches. Like the model-based approach, it does learn a transition model and reward model. Like the model-free approach, it provides a very quick decision procedure.

Rather than learn the Q function, TD learns the value function $V(s)$. The idea is quite similar to Q-learning. It is based on the fact that the value of a state s is equal to the expected immediate reward at s plus the expected future reward from the successor state s'. Every time a transition is made from s to s', we receive a sample of both the immediate and future rewards. On transitioning from s to s' and receiving reward r, the estimate for the value of s is updated according to the formula

$$V(s) \leftarrow V(s) + \alpha((r + \gamma V(s')) - V(s)) \tag{2}$$

Once again, α is the learning rate, and can be decreased as a state s is visited more often. As with Q-learning, rewards are eventually propagated backwards through the states, so that a state that leads to a good reward in the long run will have a good value associated with it.

In parallel to learning the value function, TD-value learning also learns the transition model. The value function and the transition model are used together to determine the optimal move. For example, if we are simply exploiting, then we would select action a in state s as the one that solves,

$$\pi(s) = \arg\max_{a \in \mathcal{A}} \left[R(s, a) + \sum_{s'} P(s' \mid s, a) V(s') \right]$$

In practice we would typically adopt something like ϵ-greedy exploration on top of this.

Despite the similarity of the two algorithms, TD-value learning has an advantage over Q-learning in that values are propagated more quickly. One reason is just that values are propagated on states not on (state,action) pairs and so the value of a useful action in a subsequent state will immediately propagate to earlier states. Similarly for the value of a bad action. From this, we see that the asymmetry in Q-learning disappears in the TD-value method, because the learning rule does not mention a maximization operator. A second advantage is comes directly from this being a model-based approach, in that the action that is selected using the learned model in addition to the learned value function. The learner makes better use of the information observed by the agent, and thus will typically learn more quickly.

4.2 Dyna-Q

Another interesting RL algorithm is *Dyna-Q*. This proceeds just like Q-learning except a model of the world is maintained, and additional Q update steps are done *in simulation* every time step.

As the agent has an estimate of the model, it is possible to simulate transitions from state to state. This is like interleaving a bit of planning (in this case a bit of value iteration) into Q learning. By doing so, it provides for better utilization of the information observed while acting in the world than Q-learning, and often has significantly better performance.

The one hazard is that sometimes Dyna-Q can be overly aggressive in exploiting the observations made so far in planning how to act next, and this can cause Dyna-Q to forfeit the opportunity to learn the optimal policy unless used with care. In particular, one needs to be careful to explore enough.

A slight variation on Dyna-Q, called "prioritized sweeping" is to choose the Q values to update in the planning step (the "repeat L times step") based on a priority queue of Q-updates, where prioritization is given to those for which there is a big difference between the target and the current value. As updates are made, the priority queue is maintained and "big changes" are updated first in this asynchronous manner.

5 Exploration versus Exploitation

One of the issues that makes reinforcement learning fascinating is the fact that the agent needs to make decisions as it learns. There are two basic motivations for choosing an action:

Exploitation Choose an action that leads to a good reward.

Exploration Choose an action that provides information to help to act better in the future.

For a model-based reinforcement learner, exploitation means following the optimal policy given the current estimated MDP. Similarly, for a Q-learner, exploitation means choosing the action with the highest Q value. For a TD-value learner, exploitation means choosing the action a at state s that maximizes $R(s, a) + \gamma \sum_{s'} P(s' \mid s, a)V(s')$.

Exploitation does not mean stopping learning altogether — the agent will still use the next state reached, and the reward received, to adapt its behavior in the future. However, it can mean an agent will get stuck in a local minima. Exploration means choosing which action to take based on the need to learn how to behave. There are several possibilities for which action to take in order to explore:

- Choose the action that has been tried least often in the current state.
- Choose an action that leads to states that have been unexplored.
- Choose a random action.

There is a natural tradeoff between exploration and exploitation. Exploitation helps the agent maximize its short and medium-term reward. However, in the long term, exploration is beneficial, because it can help the agent learn a better model and a better overall policy.

Suppose for example that you are learning to navigate around Boston. In the past, you have once successfully made your way from Harvard Square to Coolidge Corner in Brookline, by taking

Harvard Avenue, but found that the traffic was heavy along that route. Next time you have to get from Harvard Square to Coolidge Corner, you could simply take the same route. Or you could explore, hoping to find a better route. You expect that your explorations will hurt you in the short term. However, you believe that they might pay off in the long run, because you will discover a better route.

There is no perfect answer to the exploration-exploitation tradeoff. In general, both pure exploration and pure exploitation are bad. With pure exploration, the agent will never reap the benefits of learning. With pure exploitation, the agent will get stuck in a rut, like taking Harvard Avenue to Coolidge Corner. A mixture of both is needed. With a completely unknown model, the agent should mainly explore. As the model becomes more known, the agent should gradually shift from an exploration to an exploitation mode, though it may never completely stop exploring.

One may think that the answer is then to explore (e.g., with random actions) until some time T, and then to switch modes and exploit from then on. This is still problematic, for the following reasons:

(a) It might provide for insufficient exploration.

(b) The cost for the initial exploration phase in lost reward could be very high.

(c) It is not robust, in that if the environment changes the agent is no longer learning and not adaptive.

(d) The learning method might be "on policy" meaning that the policy learned depends on the policy taken while learning. This is not a problem for Q-learning, model-based learning, or Dyna-Q. These are all what are called "off policy" methods. But it is a problem for TD-value learning, for which the update rule converges towards the value of the policy followed.[1]

A popular and simple alternative is the ϵ-greedy exploration mentioned previously. Although useful, one problem with this approach is that we want the exploration probability to decrease as the model is better known; this can allow for learning the optimal policy in the limit. We need to ensure that in the long run two things happen:

1. Every action is taken in every state an unbounded number of times and the learning rate is eventually small enough.

2. The probability of exploration tends to zero, i.e., is zero in the limit.

A simple way to achieve theses properties was outlined above. Basically, one reduces the learning rate and the exploration rate over time in a careful way. In addition to simply decaying the exploration rate in ϵ-greedy as described above, another common method is *Boltzmann exploration*, where the probability of selecting action a in state s is given by

$$\Pr(a \mid s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}}, \tag{3}$$

[1]SARSA is another (model-free) RL method that is not discussed here and is also on-policy.

and $\tau > 0$ is the "temperature," and set to start high and decay towards zero, e.g., following an exponential cooling curve. For a high temperature this is equivalent to uniform random action selection. For a low temperature it is pure exploitation. In between it is a softmax.

6 The Curse of Dimensionality

All the methods we have looked at for reinforcement learning learn something that grows with the size of the state space. Assume an MDP with N states and M actions.

The model-based approach learns the reward model, whose size is N, and a transition model whose size is MN^2. The model learned by Q-learning is more compact, since the transition model is not learned. Nevertheless, the size of the Q function is MN. The TD-value method learns the value function, whose size is N, in addition to the transition model.[2]

We see that for all three methods, the size of the learned model is at least N. This is OK if the state space is small, but in most applications the state space is huge, and therefore learning and storing the complete model is infeasible. This is the much feared problem of the *curse of dimensionality*. A standard approach to this problem is to assume that the Q-values can themselves be approximated by solving a regression problem on a feature space. We will briefly consider the idea in the context of Q-learning, but it can just as well be used with TD-value learning.

Let $\hat{Q}_{\boldsymbol{\theta}}(s, a)$ define a parametric Q-value approximation, where $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_J)$ are the parameters to learn. For example, this could be a simple linear regression model on a set of J features, with

$$\hat{Q}_{\boldsymbol{\theta}}(s, a) = \theta_1 f_1(s, a) + \ldots + \theta_J f_J(s, a) \tag{4}$$

defined on features $f_j(s, a)$. Based on this, and recalling the basic approach of

$$\text{NewEstimate} \longleftarrow \text{OldEstimate} + \text{StepSize}[\text{Target} - \text{OldEstimate}],$$

then we can define a squared error function,

$$\text{Error}(s, a) = \frac{1}{2}\left(\hat{Q}_{\boldsymbol{\theta}}(s, a) - \text{Target}(s, a)\right)^2 \tag{5}$$

and take the derivative, with a view to doing gradient descent, with

$$\frac{\partial \text{Error}(s, a)}{\partial \theta_j} = (\hat{Q}_{\boldsymbol{\theta}}(s, a) - \text{Target}(s, a))\frac{\partial \hat{Q}_{\boldsymbol{\theta}}(s, a)}{\partial \theta_j} \tag{6}$$

Now we can adopt gradient descent, with a Q-update rule that works with this function approximation defined as

$$\theta_j \longleftarrow \theta_j + \alpha(\text{Target}(s, a) - \hat{Q}_{\boldsymbol{\theta}}(s, a))\frac{\partial \hat{Q}_{\boldsymbol{\theta}}(s, a)}{\partial \theta_j}\,. \tag{7}$$

[2]Sometimes the TD-value method can work with a more compact representation of the transition model. For example, this would be the case in Backgammon where it depends on the roll of two dice, and the transition model is already known.

This takes the place of Eq. 1, with Target$(s, a) = R(s, a) + \gamma \max_{a'} \hat{Q}_{\boldsymbol{\theta}}(s', a')$, where s' is the state reached after action a in state s. The innovation is that when a Q-update is made in state s after action a, the effect now propagates everywhere, because of the generalization that occurs because of the general representation $\hat{Q}_{\boldsymbol{\theta}}(s, a)$.

One way to think about this is that the state is actually described by a set of features (just like in supervised and unsupervised learning), $x_1, x_2, \ldots, x_J$. These variables may be binary, continuous, or take on one of a finite number of values. Any assignment $x_1, \ldots, x_J$ of values to all the variables is a possible state, so the number of states is exponential in the number of variables. Storing a complete table with the value of every (state,action) pair is infeasible. Instead, the learner tries to learn a function $\hat{Q}_{\boldsymbol{\theta}}$ on $x_1, \ldots, x_J$ and actions $\mathcal{A}$, such that $\hat{Q}_{\boldsymbol{\theta}}(x_1, \ldots, x_J, a)$ is approximately the Q-value. This technique is called *Q-value function approximation* because the learner learns a compact approximation to the Q-value, rather than storing it exactly.

As we know by now, the problem of learning a function from a set of variables to a value is a typical problem in supervised machine learning. It is possible, for example, to use neural networks or SVMs or linear regression to learn this function. Practical applications of reinforcement learning generally integrate generalization methods such as this into the RL framework. The basic method that is adopted is gradient descent in parameter space, with this step used whenever new information is received following taking an action in some state.

7 Applications of RL

Reinforcement learning is a very old idea in AI. It goes back to the very first years of AI — 1959, in fact, and Arthur Samuel's checkers playing program.

7.1 Checkers

Samuel's program used a simple form of value function approximation. In his approach, the state of a game of checkers is represented by a set of variables. For example, the state variables may be [3]

x_1	Number of black pieces
x_2	Number of red pieces
x_3	Number of black kings
x_4	Number of red kings
x_5	Number of black pieces threatened by red
x_6	Number of red pieces threatened by black

Samuel assumed that the value function was a linear function of these variables. That is, that there exist weights $w_0, \ldots, w_6$ such that

$$V(x_1, x_2, x_3, x_4, x_5, x_6) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + w_5 x_5 + w_6 x_6$$

[3] This formulation is from Chapter 1 of Mitchell, "Machine Learning".

Whenever the state of the game changed from s to s', the program used $V(s')$, the value at the successor state, as an estimate for $V(s)$, the value at the previous state, like in TD-value learning. He used the new estimate to adjust the weights as follows:

$$w_j \leftarrow w_j + \alpha(V(s') - V(s))x_j \quad (x_0 \text{ is always } 1)$$

This is the standard rule for minimizing squared error via gradient descent. Notice that the degree to which the weights need to be adjusted in general is proportional to the error $V(s') - V(s)$. The degree to which the particular weight w_j needs to be adjusted is proportional to x_j, because x_j determines what contribution this particular weight made to the error.

Samuel trained his program by having it play many games against itself. Samuel was not an expert, and the program learned to play much better than he did. Thus, already in 1959, Samuel refuted the charge that AI is impossible because "a computer can only do what it is programmed to do". By using learning, Samuel's checkers player was able to play a much better game than Samuel could have told it to play.

7.2 Backgammon

Gerry Tesauro used a similar approach in developing his TD-Gammon backgammon player. As you can tell by the name, TD-Gammon is based on TD-value learning. It also uses value function approximation, using a neural network to estimate the value of a state of the game. Before developing TD-Gammon, Tesauro had built the Neurogammon player, which used supervised learning rather than reinforcement learning to learn the Q function. That is, the program was given a set of positions labeled by a human backgammon expert, and tried to learn the Q function directly from those. Neurogammon was a decent backgammon player, but not world-class. In contrast, TD-Gammon learned by playing against itself, and the only feedback it received was whether or not it won the game. TD-Gammon became a much better player than Neurogammon, and reached world-champion level.

7.3 Control

Both checkers and backgammon are examples of domains where the transition model is known, but the state space is too large to enumerate. The success of reinforcement learning in these domains shows that it is a viable alternative to computing optimal policies directly.

Reinforcement learning has also been applied to a variety of domains in robot navigation and control. One example is the "inverted pendulum" problem. There is a cart that can move left or right on a track. On the cart is balanced a pole. The goal is to control the position of the cart so as to keep the pole balanced upright, while keeping the cart on the track. A very early success came in 1968, when Michie and Chambers built a system using reinforcement learning that learned how to balance the pole for over an hour. Their work led to a spate of further research on this and similar problems. More recently, RL techniques have been demonstrated by Andrew Ng and collaborators to be useful for the control of helicopter flight.

One interesting idea, proposed by Rich Sutton, has emerged recently in the reinforcement learning community. That is that reinforcement learning *can be used to learn concepts as well as* *policies*. Imagine that you have a robot navigation problem where you need to make your way from one room to another via one of two doors. Reinforcement learning can learn a policy that will tell you which door to head for from any point in the room. This policy will divide the room into two regions: those from which the robot should head for the left hand door, and those from which it should head for the right hand door. A sophisticated robot could interpret the results as meaning that the room is divided into two parts, a left-hand side and a right-hand side. This distinction is based solely on the relationship of each location to the goal, and not on any actual features of the locations. Furthermore, the only feedback the robot ever gets is rewards for reaching the goal. The fact that it can learn to make the distinction is quite remarkable, and lends plausibility to the claim that reinforcement learning could be a foundation for intelligence. Whether or not you agree with this claim, reinforcement learning is one of the most fascinating ideas around in artificial intelligence.

Changelog

- 23 November 2018 – Initial version converted from Harvard CS181 course notes.