Reinforcement Learning

Michael Rose

In which we examine how an agent can learn from success and failure, from reward and punishment.

21.1 | Introduction

In this chapter we will study how agents can learn what to do in the absence of labeled examples of what to do. The agent needs to know whether something good or something bad has happened. This is called **reward** or **reinforcement**. The reward can happen infrequently (such as at the end of a chess game) or frequently (such as ping pong).

We will look at 3 different agent designs:

- A **utility based agent** learns a utility function on states and uses it to select actions that maximize the expected outcome utility
- A utility based agent must have a model of the environment in order to make decisions, because it must know the states to which its actions will lead.
- A **Q learning agent** learns an **action-utility function**, or **Q-function**, giving the expected utility of taking a given action in a given state.
- A Q learning agent can compare the expected utilities for its available choices without needing to know their outcomes, so it doesn't need a model of the environment.
- A **reflex agent** learns a policy that maps directly from states to actions.

21.2 | Passive Reinforcement Learning

Suppose we have a passive learning agent using a state based representation in a fully observable environment. In passive learning, the agent's policy π is fixed: in state s, it always executes the action $\pi(s)$. Its goal is to learn how good the policy is, through learning the utility function $U^{\pi}(s)$.

The passive learning task is similar to the **policy evaluation** task, part of the **policy iteration algorithm** described in 17.3. The main difference is that the passive learning agent does not know the **transition model** $P(s'|s,a)$, which specifies the probability of reaching state s' from s after doing action a. It also doesn't know the **reward function** $R(s)$, which specifies the reward for each state.

The agent executes a set of **trials** in the environment using policy π. The objective is to use the information about rewards to learn the expected utility $U^{\pi}(s)$ associated with each nonterminal state s. The utility is defined to be the expected sum of discounted rewards obtained if policy π is followed:

$$U^{\pi}(s) = \mathrm{E}[\sum_{t=0}^{\infty} \gamma^t R(S_t)]$$

where $R(s)$ is the reward for a state, S_t (a random variable) is the state reached at time t when executing policy π, $S_0 = s$, and γ is a discount factor, generally set to 1.

21.2.1 | Direct Utility Estimation

Direct utility estimation is a method from the area of **adaptive control theory** in which the idea is that the utility of a state is the expected total reward from that state onward (called the expected **reward-to-go**), and each trial provides a sample of this quantity for each state visited. For each trial, a sample total reward is calculated. Then, at the end of each sequence, the algorithm calculates the observed reward to go for each state and updates the estimated utility for each state in a table. In the limit of infinitely many trials, the sample average will converge to its true expectation.

Since direct utility estimation is an instance of supervised learning where each example has state as input and the observed reward to go as output, we have essentially reduced reinforcement learning to a standard inductive learning problem. Unfortunately, it misses something important: The utilities of states are not independent. The utility of each state equals its own reward plus the expected utility of its successor states. That is, the utility values obey the Bellman equations for a fixed policy:

$$U^\pi(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) U^\pi(s')$$

More broadly, we can view direct utility estimation as searching for U in a hypothesis space that is much larger than it needs to be, in that it holds many functions that violate the Bellman equations. For this reason, it often converges very slowly.

21.2.2 | Adaptive Dynamic Programming

An **adaptive dynamic programming agent** takes advantages of the constraints among the utilities of states by learning the transition model that connects them and solving the corresponding Markov decision process using a dynamic programming method. For a passive learning agent, this means plugging the learned transition model $P(s'|s, \pi(s))$ and the observed rewards $R(s)$ into the Bellman equations to calculate the utilities of states. As these equations are linear, they can be solved with any linear algebra library.

Alternatively, we could use the approach of **modified policy iteration**, which uses a simplified value iteration process to update the utility estimates after each change to the learned model. Since the model changes only slightly with each observation, the value iteration process can use the previous utility estimates as initial values and should converge quite quickly.

Generally, we can use the ADP agent to provide a measurement standard with which to compare for other reinforcement algorithms. It is, however, intractable for large state spaces.

ADP uses maximum likelihood estimation to learn the transition model. By choosing a policy solely based on the estimated model, it is acting a if the model were correct. This is not necessarily a good idea. Instead, it might be a good idea to choose a policy that, while not optimal for the model estimated by maximum likelihood, works reasonably well for the whole range of models that have a reasonable chance of being the true model. There are two mathematical approaches that have this flavor:

Bayesian Reinforcement Learning assumes a prior probability $P(h)$ for each hypothesis h about what the true model is. Then, with Bayes rule we obtain the posterior $P(h|e)$. Then, if the aget has decided to stop learning, the optimal policy is the one that gives the highest expected utility.

Let u_h^π be the expected utility, averaged over all possible start states, obtained by executing policy π in model h. Then we have

$$\pi^* = \arg\max_\pi \sum_h P(h|e) u_h^\pi$$

Robust Control Theory allows for a set of possible models $\mathcal{H}$ and defines an optimal robust policy as one that gives the best outcome in the worst case over $\mathcal{H}$:

$$\pi^* = \arg\max_{\pi} \min_{h} u_h^{\pi}$$

Often, the set $\mathcal{H}$ will be the set of models that exceed some likelihood threshold on $P(h|e)$, so the robust and Bayesian approaches are related. Sometimes the robust solution can be computed efficiently. There are many other reinforcement algorithms that produce robust solutions as well.

21.2.3 | Temporal Difference Learning

Solving the underlying Markov decision process as in the preceding section is not the only way to bring the Bellman equations to bear on the learning problem. Another way is to use the observed transitions to adjust the utilities of the observed states so that they can agree with the constraint equations. More generally, when a transition occurs from state s to s', we apply the following update to $U^{\pi}(s)$:

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s))$$

where α is the learning rate parameter. Since this update rule uses the difference in utilities between successive states, it is often called the **temporal difference equation**.

All temporal difference methods work by adjusting the utility estimates towards the ideal equilibrium that hold locally when the utility estimates are correct. Note that TD doesn't need a transition model to perform its updates. The environment supplies the connection between neighboring states in the form of observed transitions.

The ADP and TD approaches are closely related. Both try to make local adjustments to utility estimates in order to make each state agree with its successors. One difference is that TD adjusts a state to agree with its observed succesor, whereas ADP adjusts the state to agree with all of the successors that might occur, weighted by their probabilities. This difference disappears when the effects of TD adjustments are averaged over a large number of transitions, because the frequency of each succesor in the set of transitions is approximately proportional to its probability.

Another difference is that whereas TD makes a single adjustment per observed transition, ADP makes as many as it needs to restore consistency between the utility estimates U and the environment model P. Although the observed transition makes only a local change in P, its effects might need to be propagated throughout U. Thus, TD can be viewed as a crude but efficient first approximation to ADP.

We can make more efficient versions of ADP by directly approximating the algorithms for value iteration or policy iteration. Even though the value iteration algorithm is efficient, it is intractable for large state spaces. Since each of the necessary adjustments to state space are tiny, we can generate reasonably good answers* quickly by bounding the number of adjustments made after each observed transition.

We can also use a heuristic to rank the possible adjustments so as to carry out only the most significant ones. The **prioritized sweeping** heuristic prefers to make afjustments to states whose likely successors have just undergone a large adjustment in their own utility estimates.

Using adjustments like this, we can use approximate ADP to have state spaces several magnitudes larger than the original.

21.3 | Active Reinforcement Learning

A passive learning agent has a fixed policy that determines its behavior. An active agent must decide what actions to take.

21.3.1 | Exploration

Following the optimal policy for the learned model at each step is a **greedy** approach. Repeated experiments show that the greedy agent very seldom converges to the optimal policy for the environment and sometimes converges to very bad policies. What the greedy agent overlooks is that actions do more than provide rewards according to the current learned model; they also contibute to learning the true model by affecting the percepts that are received. By improving the model, the agent will receive greater rewards in the future. An agent therefore must make a tradeoff between **exploitation** to maximize its reward, and **exploration** to maximize its long term wellbeing. Pure exploitation risks getting stuck in a rut, and pure exploration to improve ones knowledge is of no use if one never puts that knowledge into practice.

Can we find an optimal exploration policy that strikes the right balance between exploration and exploitation? This question is studied in depth in the subfield of statistical decision theory that deals with **bandit problems**. Although bandit problems are difficult to solve to obtain an optimal exploration method, it is possible to come up with a reasonable scheme that will eventually lead to optimal behavior by an agent.

Technically, any such scheme must be greedy in the limit of infinite exploration, or **GLIE**. A GLIE scheme must try each action in each state an unbounded number of times to avoid having a finite probability that an optimal action is missed because of an unusually bad series of outcomes.

There are several GLIE schemes. One of the simplest is to have the agent choose a random action a fraction $1/t$ of the time and to follow the greedy policy otherwise. While this converges, it does so slowly.

A more sensible approach is to give some weight to actions that the agent has not tried very often, while tending to avoid actions that are believed to be of low utility. This can be implemented by altering the constraint equation such that it assigns a higher utility estimate to relatively unexplored state action pairs. This amounts to an optimistic prior over the possible environments and causes the agent to believe initially as if there were wonderful rewards scattered all over the place.

Let $U^+(s)$ denote the optimistic estimate of the utility of state s, and let $N(s,a)$ be the number of times action a has been tried in state s. Then we can rewrite the update equation for value iteration in an ADP learning agent as

$$U^+(s) \leftarrow R(s) + \gamma \max_a f(\sum_{s'} P(s'|s,a)U^+(s'), N(s,a))$$

where $f(u,n)$ is called the **exploration function**. It determines how greed is traded off against curiousity.

21.3.2 | Learning an Action-Utility Function

Now that we have an active ADP agent, we seek to construct an active temporal difference learning agent. To first change from the passive case is that the agent is no longer equipped with a fixed policy, so if it learns a utility function U, it will need to learn a model in order to be able to choose an action based on U via one step look ahead. The TD update rule stays the same.

There is an alternative TD method, called **Q-learning**, which leans an action-utility representation instead of learning utilities. Let $Q(s,a)$ denote the value of doing action a in state s. Q-values are directly related to utility values as follows:

$$U(s) = \max_a Q(s,a)$$

A TD agent that learns a Q-function does not need a model of the form $P(s'|s,a)$, either for learning or for action selection. For this reason, Q-learning is called a **model-free** method. The constraint equation is

$$Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a)$$

We can use this equation directly as an update equation for an iteration process that calculates exact Q-values, given an estimated model. This does require that a model also be learned, since the equation uses $P(s'|s,a)$.

The temporal-difference approach requires no model of state transitions - all it needs are the Q values. Then the update equation for TD Q-learning is

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

which is calculated whenever action a is executed in state s leading to state s'.

Q-learning has a close relative called **SARSA** (State-Action-Reward-State-Action). The update rule for SARSA is

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma Q(s',a') - Q(s,a))$$

where a' is the action actually taken in state s'. The rule is applied at the end of each s, a, r, s', a' quintuplet.

The difference from Q learning is subtle. Whereas Q-learning backs up the best Q-value from the state reached in the observed transition, SARSA waits until an action is actually taken and backs up the Q value for that action. Q-learning is more flexible than SARSA, in the sense that a Q-learning agent can learn how to behave well even when guided by a random or adversarial exploration policy.

21.4 | Generalization in Reinforcement Learning