CS181 Lecture 5 — Reinforcement Learning

The topic for today is reinforcement learning. We will look at several methods, and then 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 from the environment, 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 the Markov Decision Process, and its extension, the Partially Observable Markov Decision Process. We considered methods 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. We will, however, assume that the agent does know the current state of the world when it gets to make a decision.[1] The interaction story is similar to 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 the model of the world, and so obviously it cannot compute the optimal

policy.

However, one might hope that the agent would be able to learn a good policy, based on the rewards and punishments that it gets. 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 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. We're going to show that reinforcement learning has a computational basis, and that agents can perform quite well using reinforcement

[1]Unknown state caused enough problems when the model was known; the combination of unknown state and unknown model is difficult.

learning. In fact, there have been some remarkable successes. For example, a world-class backgammon player is based on reinforcement learning.

There are two things that makes 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 move that really led to your winning the game. 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 in the learning process. 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 the exploration-exploitation tradeoff .

We shall discuss three algorithms for reinforcement learning: learning the

model, Q learning, and temporal difference learning.

2 Learning the Model

The first approach to reinforcement learning is simple: just learn the transition and reward models. 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, in which the expected reward in state i is x, with a deterministic model in which the reward in state i is x.

In our situation, we are trying to learn the 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 $N_a{}_i$ of the number of times we have visited state i and taken action a, and $R_a{}_i$, the total reward accumulated in all times action a was taken in state i. Our estimate for the expected reward for action a in state i is simply $R_a{}_i/N_a{}_i$.

As for the transition model, that is also very simple to learn. For each state i and action a, we need to learn $T_a{}_i$, the probability distribution over states reached by taking action a in state i. To do this, we store $N_a{}_i$, the total number of times action a was taken in state i, and $N_a{}_{ij}$, the number of times a transition from i to j happened on action a. The estimate for $T_a{}_{ij}$ is simply $N_a{}_{ij}/N_a{}_i$. (This notation, using N with superscripts or subscripts to maintain counts, will be used frequently during the course. The counts that are used to estimate a value of interest are called sufficient statistics.)

Once estimates for the reward and transition models have been made, the agent can solve the model to determine an optimal policy. 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.

One issue to address using the model-based approach is how often the agent should solve the MDP and update its policy. At one extreme, the agent could update its estimates of the transition and reward model every time it takes an action, and immediately solve the updated MDP. Alternatively, the agent might take time to recompute its policy after it has taken a large number of actions. Consider that if the agent has been learning for a while, the optimal value function and optimal policy are unlikely to change much after one action. On the one hand, this would argue that the agent should not waste its time recomputing its policy at every iteration. On the other hand, one can use this fact to greatly speed up the online solution process. 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 algorithm will converge in one iteration.

3 Q Learning

The second approach to reinforcement learning dispenses with learning a model, and tries to learn the value of taking each action in each state. This approach is called model-free, in contrast to the model-based approach described above. Recall that for the purpose of solving MDPs we defined the function Q(i, a) which provides the expected value of taking action a in state i. The Q value incorporates both the immediate reward and expected value of the next state

reached. [2] A Q function also determines a policy and a value function: $\pi_Q(i) = \arg\max_a Q(i, a)$ and $V_Q(i) = \max_a Q(i, a)$. The Q learning algorithm tries to

learn the Q function directly. Once an agent has learned a Q function, it will choose the action a at each state i that maximizes Q(a, i).

How does the agent learn the Q function? Expanding the definition of the Q function, and using the policy determined by the Q function to expand the value function, we have

$$\begin{aligned} Q(i, a) &= R(i, a) + \sum_j T^a_{ij} V_Q(j) \\ &= R(i, a) + \sum_j T^a_{ij} \max_b Q(j, b) \\ &= \sum_j T^a_{ij} [R(i, a) + \max_b Q(j, b)] \quad (*) \end{aligned}$$

Formula (*) follows because $\sum_j T^a_{ij} = 1$. Now, the agent does not know T^a_{ij}, and because this is a model-free approach it does not even try to learn it. But look at the right hand side of Formula (*). It is the sum, over possible next states j, of the probability of reaching j, times a term that depends on j. In other words, it is an expectation, where the expectation is taken over the distribution T^a_i.

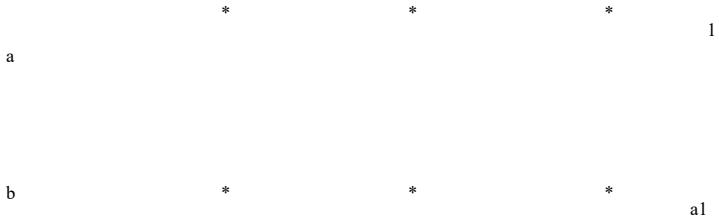
Our goal then, is to estimate Q(i, a) as the expectation, where j is drawn from T^a_i, of $R(i, a)+\max_b Q(j, b)$. Each time we actually take a transition under action a from state i to some other state j, receiving an actual reward r, we get a sample for estimating the expectation. We can use this sample for updating our old estimate of Q(i, a). Specifically, on transitioning from i to j under action a and receiving reward r, the following formula is used:

$$Q(i, a) \leftarrow (1 - \alpha)Q(i, a) + \alpha(r + \max_b Q(j, b)) \quad (1)$$

The interpretation of this formula is that to get the new estimate of Q(i, a), we mix in a little bit of the new sample into the old estimate. The parameter α is called the learning rate. It determines how much of an effect the new sample has on the estimate. If α is large, learning will be quick, but the estimate of Q(i, a) will be subject to wild fluctuations. If α is small, the estimate will only change slightly when a new sample is obtained, and learning will be slow. A natural thing to do is to decrease α gradually as the number of samples of Q(i, a) increases.

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:

[2]I am ignoring discounting in the presentation of reinforcement learning. If you want to use total discounted reward as the objective function, multiply terms representing the expected future reward by the discount factor γ wherever they appear.



Here the critical decision happens early: choosing a at the first step eventually leads to a reward of 1, while b 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. It turns out that the number of runs required for positive rewards to filter backwards is linear in the number of steps, but for negative rewards it is exponential.

This example suggests an optimization to the Q learning algorithm. For a game-like environment, in which a score is received at the end of the game that may be a result of decisions made early in the game, a good idea is to wait until the end of the game to perform the learning. At that point, the learning is done from the end of the game backwards. This allows the result of the game to be propagated backwards to all the nodes in that game.

4 Temporal Difference Learning

There are good arguments for both model-based and model-free approaches. The model-free approach is very simple. It is also psychologically appealing. 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 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 really 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?

The temporal difference method (TD) 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. Like the model-free approach, it provides a very quick decision procedure. TD also makes sense if the transition model is known, but the MDP is to hard to solve.

Rather than learn the Q function, TD learns the value function V (i). The idea is quite similar to Q learning. It is based on the fact that the value of a state i is equal to the expected immediate reward at i plus the expected future reward from the successor state j. Every time a transition is made from i to j, we receive a sample of both the immediate and future rewards. On transitioning

from i to j and receiving reward r, the estimate for the value of i is updated according to the formula

$$V(i) \leftarrow (1 - \alpha)V(i) + \alpha(r + V(j)) \tag{2}$$

Once again, α is the learning rate, and can be decreased as the state i 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. The same optimization of waiting until the end of the game and then propagating backwards can be performed for TD learning as for Q learning.

In parallel to learning the value function, TD learning also learns the transition model. Alternatively, the transition model may already be known, but the MDP be too hard to solve. The value function and the transition model are used together to determine the optimal move. The optimal action a to make in state i is the one that maximizes $R(i, a) + \sum_j T^a_{ij} V(j)$.

Despite the similarity of the two algorithms, temporal difference learning has an advantage over Q learning in that values are propagated more quickly. There is an asymmetry in Q learning as a result of the maximization operator in the learning rule. This asymmetry leads to the result that while rewards are propagated backwards reasonably quickly, punishments are propagated much more slowly. A punishment will only be propagated back from a state j to a previous state i only if it is unavoidable that the punishment will be received once j is reached — i.e., only if it shows up in Q(j, a) for every action a. Even if rewards can be propagated backwards in a single run (using the optimization of waiting till the end of the run), it can take many runs to discover that a punishment is unavoidable. TD learning does not suffer from this disadvantage, because the learning rule does not mention a maximization operator.

5 Exploration vs Exploitation

One of the issues that makes reinforcement learning fascinating is the fact that you have to make decisions and take actions even as you are learning. But if you have not yet learned, or only partially learned, how do you decide what action to take? There are two basic motivations for choosing an action:

Exploitation Choose an action that leads to a good reward.

Exploration Choose an action that helps you to learn better.

For a model-based reinforcement learner, exploitation means solving the current estimated MDP in order to decide what to do next. Similarly, for a Q learner, exploitation means choosing the action at a state with the highest Q value. For a TD learner, exploitation means choosing the action a at state i that maximizes $R(i, a) + \sum_j T_{ij}^a V(j)$.

Exploitation does not mean stopping learning altogether — the agent will still use the next state reached and the reward received to update its estimate

of the MDP. However, it does mean that its consideration in determining what to do is performance, and not improving its model.

Exploration means choosing which action to take based solely on the need to learn the model. 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 until some time T, and then to switch modes and exploit from then on. This is still problematic, for two reasons. The first reason is that if you only explore for a finite amount of time, your exploration may be insufficient to discover the optimal policy. In order to guarantee convergence to the optimal policy in the long run, you need to be able to explore arbitrarily often.

The second reason that switching from pure exploration to pure exploitation is a bad idea is that the values of states depends on the policy you're going to use. Some exploitation is a good idea even if your only goal is to learn, because exploitation produces more accurate value and Q functions. You want to learn the values relative to the actual exploiting policy, not a random exploring policy. Consider again the example of learning to navigate around Boston. If you only explore, you will essentially drive around randomly. You will never (or take ages to) learn to take an action that is only good because it can be exploited later. For example, you will not learn to take the freeway to get from A to B, because getting on the freeway is only beneficial if you get off at the right exit. If you

overshoot the exit it is very costly. So if you are moving about randomly you will find that getting on the freeway is almost always a bad idea! In order to learn that the freeway is a good idea, you must explore a little so that you get on it, but you must also exploit so that you get off at the right place. Therefore, some degree of exploitation is needed even while learning.

Another simple strategy is to fix some probability p, such that at each move, the agent explores with probability p, and exploits with probability $1 - p$. This is also not a great idea. Intuitively, we want the exploration probability to decrease as the model is better known. There is also a theoretical reason that a fixed probability is a bad idea. If we want to guarantee that in the long run, the expected utility earned by the agent will converge to the maximal expected utility for the MDP, we need to make sure that in the long run, the probability of exploration tends to zero. Basically, we need to ensure that in the long run, two things happen:

1. Every action is taken in every state infinitely often.

2. The probability of exploration tends to zero.

There are various ways to implement an exploration/exploitation policy that

has these properties. For example, the probability of exploration may depend on the total time that has passed, or on the number of times the current state has been visited. The probability may decay linearly or exponentially.

6 Generalization

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 the 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 temporal-difference method learns the value function, whose size is n, in addition to the transition model. However, in practice, the TD method does not need to store the transition model explicitly, since it only uses it to determine which next state to go to from any particular state. All that is needed is to be able to compute the probability distribution over the next state j, for the particular state i that has already been reached.

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. The standard approach to this problem is to generalize: to assume that there are some general properties of states that determine their values, or Q values. I will describe the generalization approach in the context of TD learning, but it can just as well be used with Q learning.

A very large state space is described by a set of variables $X_1,...,X_n$. These variables may be boolean, continuous, or take on one of a finite number of values. Any assignment $< x_1,...,x_n >$ 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 is infeasible. Instead, the learner tries to learn a function f on $X_1,...,X_n$, such that $f<x_1,...,x_n>$ is approximately the value of the state $< x_1,...,x_n >$. This technique is called value function approximation because the learner learns a compact approximation to the value function, rather than storing it exactly.

The problem of learning a function from a set of variables to a value is a typical machine learning problem. We will show later how neural networks can be used to encode and learn such a function. So we will postpone a full discussion

of how to incorporate generalization into reinforcement learning until after we have studied neural networks. Suffice it to say that practical applications of reinforcement learning generally integrate it with other types of learning.

7 Applications

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. 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. I.e., that there exist weights $w_0,...,w_6$ such that

$$V(< x_1,x_2,x_3,x_4,x_5,x_6 >) = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6$$

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

$$w_i \leftarrow w_i + \alpha(V(j) - V(i))x_i \qquad (x_0 \text{ is always } 1)$$

We will see later how to justify this training rule. For now, notice the following: the degree to which the weights need to be adjusted in general is proportional to the error V (j) − V (i). The degree to which the particular weight w_i needs to be adjusted is proportional to x_i, because x_i 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

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

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.

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 temporal difference 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.

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. In 1968, 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.

One very 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.