CHAPTER 11

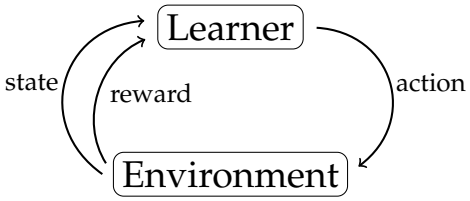
Reinforcement learning

So far, all of the learning problems we have looked at have been either *unsupervised*, in that we are given data and no expected outputs, or *supervised*, that is, for each training input $x^{(i)}$, we are told which value $y^{(i)}$ should be the output. *Reinforcement learning* differs from previous learning problems in several important ways:

- The learner interacts explicitly with an environment, rather than implicitly as in supervised learning (through an available training data set of $(x^{(i)}, y^{(i)})$ pairs drawn from the environment).
- The learner has some choice over what new information it seeks to gain from the environment.
- The learner updates models incrementally as additional information about the environment becomes available.

Online learning is a variant of supervised learning in which new data pairs become available over time and the model is updated, e.g., by retraining over the entire larger data set, or by weight update using just the new data.

In a reinforcement learning problem, the interaction with the environment takes a particular form:



- Learner observes *input* state $s^{(i)}$
- Learner generates *output* action $a^{(i)}$
- Learner observes *reward* $r^{(i)}$
- Learner observes *input* state $s^{(i+1)}$
- Learner generates *output* action $a^{(i+1)}$
- Learner observes *reward* $r^{(i+1)}$
- ...

The learner is supposed to find a *policy*, mapping state s to action a, that maximizes expected reward over time.

11.1 Reinforcement learning algorithms overview

A *reinforcement-learning (RL) algorithm* is a kind of a policy that depends on the whole history of states, actions, and rewards and selects the next action to take. There are several different ways to measure the quality of an RL algorithm, including:

- Ignoring the $r^{(i)}$ values that it gets *while* learning, but consider how many interactions with the environment are required for it to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$ that is nearly optimal.
- Maximizing the expected discounted sum of total rewards while it is learning.

Most of the focus is on the first criterion, because the second one is very difficult. The first criterion is reasonable when the learning can take place somewhere safe (imagine a robot learning, inside the robot factory, where it can't hurt itself too badly) or in a simulated environment.

Approaches to reinforcement-learning differ significantly according to what kind of hypothesis or model they learn. Roughly speaking, RL methods can be categorized into model-free methods and model-based methods. The main distinction lies in whether we are explicitly learning the transition and reward models to assist the end-goal of learning a policy. We will start our discussion with the model-free methods, and introduce two of the arguably most popular types of algorithms, Q-learning (Section 11.2.1) and policy search (Section 11.2.4). We then describe model-based methods (Section 11.3). Finally, we consider "bandit" problems (Section 11.4), which differ from our MDP learning context by having probabilistic rewards.

11.2 Model-free methods

Model-free methods are methods that do not explicitly learn transition and rewards models. Depending on what is explicitly being learned, model-free methods are sometimes further categorized into value-based methods (where value is a shorthand for value functions) and policy-based methods. It's important to note that such categorization is approximate and the boundaries are blurry. In fact, current RL research tends to combine the learning of value functions, policies, and transition and reward models all into a complex learning algorithm, in an attempt to combine the strengths of each approach.

11.2.1 Q-learning

Q-learning is a frequently used class of RL algorithms that concentrates on learning (estimating) the state-action value function, the Q function. Specifically, recall the MDP value-iteration update:

$$Q(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(s', a') \tag{11.1}$$

The thing that most students seem to get confused about is when we do value iteration and when we do Q learning. Value iteration assumes you know T and R and just need to *compute* Q. In Q learning, we don't know or even directly estimate T and R: we estimate Q directly from experience!

The Q-learning algorithm below adapts this value-iteration idea to the RL scenario, where we do not know the transition function T or reward function R, and relies on samples to perform the updates.

Q-LEARNING($\mathcal{S}, \mathcal{A}, s_0, \gamma, \alpha$)

for $s \in \mathcal{S}, a \in \mathcal{A}$:
    $Q[s, a] = 0$
$s = s_0$ // (e.g., s_0 can be drawn randomly from $\mathcal{S}$)
while True:
    $a =$ select_action(s, Q)
    $r, s' =$ execute(a)
    $Q[s, a] \leftarrow (1 - \alpha)Q[s, a] + \alpha(r + \gamma \max_{a'} Q[s', a'])$
    $s \leftarrow s'$

Here, α represents the "learning rate," which needs to decay for convergence purposes, but in practice is often set to a constant. It's also worth mentioning that Q-learning assumes a discrete state and action space where states and actions take on discrete values like $1, 2, 3, \ldots$ etc. In contrast, a continuous state space would allow the state to take values from, say, a continuous range of numbers; for example, the state could be any real number in the interval $[1, 3]$. Similarly, a continuous action space would allow the action to be drawn from, e.g., a continuous range of numbers. There are now many extensions developed based on Q-learning that can handle continuous state and action spaces (we'll look at one soon), and therefore the algorithm above is also sometimes referred to more verbosely or specifically as tabular Q-learning.

In the Q-learning update rule

$$Q[s, a] \leftarrow (1 - \alpha)Q[s, a] + \alpha(r + \gamma \max_{a'} Q[s', a']) \tag{11.2}$$

the term $r + \gamma \max_{a'} Q[s', a']$ is often referred to as the (one-step look-ahead) *target*. The update can be viewed as a combination of two different iterative processes that we have already seen: the combination of an old estimate with the target using a running average with a learning rate α, and the dynamic-programming update of a Q value from value iteration.

Eq. 11.2 can also be equivalently rewritten as

$$Q[s, a] \leftarrow Q[s, a] + \alpha \left((r + \gamma \max_{a'} Q[s', a']) - Q[s, a] \right), \tag{11.3}$$

which allows us to interpret Q-learning in yet another way: we make an update (or correction) based on the temporal difference between the target and the current estimated value $Q[s, a]$.

The Q-learning algorithm above includes a procedure called *select_action*, that, given the current state s, has to decide which action to take. If the Q value is estimated very accurately and the agent is behaving in the world, then generally we would want to choose the apparently optimal action $\arg\max_{a \in \mathcal{A}} Q(s, a)$. But, during learning, the Q value estimates won't be very good and exploration is important. However, exploring completely at random is also usually not the best strategy while learning, because it is good to focus your attention on the parts of the state space that are likely to be visited when executing a good policy (not a bad or random one).

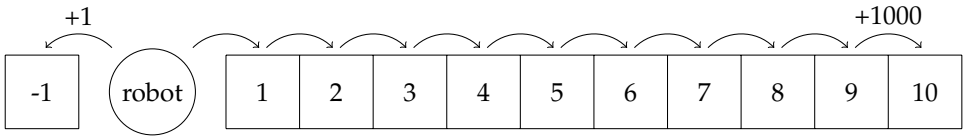
A typical action-selection strategy that attempts at addressing this *exploration versus exploitation* dilemma is the so-called ϵ-*greedy* strategy:

- with probability $1 - \epsilon$, choose $\arg\max_{a \in \mathcal{A}} Q(s, a)$;
- with probability ϵ, choose the action $a \in \mathcal{A}$ uniformly at random.

where the ϵ probability of choosing a random action helps the agent to explore and try out actions that might not seem so desirable at the moment.

Q-learning has the surprising property that it is *guaranteed* to converge to the actual optimal Q function under fairly weak conditions! Any exploration strategy is okay as long as it tries every action infinitely often on an infinite run (so that it doesn't converge prematurely to a bad action choice).

Q-learning can be very inefficient: imagine a robot that has a choice between moving to the left and getting a reward of 1, then returning to its initial state, or moving to the right and walking down a 10-step hallway in order to get a reward of 1000, then returning to its initial state.



The first time the robot moves to the right and goes down the hallway, it will update the Q value just for state 9 on the hallway and action "right" to have a high value, but it won't yet understand that moving to the right in the earlier steps was a good choice. The next time it moves down the hallway it updates the value of the state before the last one, and so on. After 10 trips down the hallway, it now can see that it is better to move to the right than to the left.

More concretely, consider the vector of Q values $Q(i = 0, \ldots, 9; \text{right})$, representing the Q values for moving right at each of the positions $i = 0, \ldots, 9$. Position index 0 is the starting position of the robot as pictured above.

Then, for $\alpha = 1$ and $\gamma = 0.9$, Eq. 11.3 becomes

$$Q(i, \text{ right}) = R(i, \text{ right}) + 0.9 \cdot \max_a Q(i+1, a). \tag{11.4}$$

Starting with Q values of 0,

$$Q^{(0)}(i = 0, \ldots, 9; \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{11.5}$$

Since the only nonzero reward from moving right is $R(9, \text{ right}) = 1000$, after our robot makes it down the hallway once, our new Q vector is

$$Q^{(1)}(i = 0, \ldots, 9; \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1000 \end{bmatrix}. \tag{11.6}$$

We are violating our usual notational conventions here, and writing $Q^{(i)}$ to mean the Q value function that results after the robot runs all the way to the end of the hallway, when executing the policy that always moves to the right.

After making its way down the hallway again, $Q(8, \text{ right}) = 0 + 0.9 \cdot Q(9, \text{ right}) = 900$ updates:

$$Q^{(2)}(i = 0, \ldots, 9; \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 900 & 1000 \end{bmatrix}. \tag{11.7}$$

Similarly,

$$Q^{(3)}(i = 0, \ldots, 9; \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 810 & 900 & 1000 \end{bmatrix} \tag{11.8}$$

$$Q^{(4)}(i = 0, \ldots, 9; \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 729 & 810 & 900 & 1000 \end{bmatrix} \tag{11.9}$$

$$\vdots \tag{11.10}$$

$$Q^{(10)}(i = 0, \ldots, 9; \text{ right}) = \begin{bmatrix} 387.4 & 420.5 & 478.3 & 531.4 & 590.5 & 656.1 & 729 & 810 & 900 & 1000 \end{bmatrix}, \tag{11.11}$$

and the robot finally sees the value of moving right from position 0.

We can see how the example above interacts with the exploration/-exploitation dilemma: from the perspective of $s_0 = 0$, it will seem, for a long time, that getting the immediate reward of 1 is a better idea, and it would be easy to converge on that as a strategy without exploring

Study Question: Determine the Q value functions that will result from updates due to the robot always executing the "move left" policy.

11.2.2 Function approximation: Deep Q learning

In our Q-learning algorithm above, we essentially keep track of each Q value in a table, indexed by s and a. What do we do if $\mathcal{S}$ and/or $\mathcal{A}$ are large (or continuous)?

We can use a function approximator like a neural network to store Q values. For example, we could design a neural network that takes in inputs s and a, and outputs $Q(s, a)$. We can treat this as a regression problem, optimizing this loss:

$$\left(Q(s, a) - (r + \gamma \max_{a'} Q(s', a'))\right)^2 , \tag{11.12}$$

This is the so-called squared Bellman error; as the name suggests, it's closely related to the Bellman equation we saw in MDPs in Chapter 10. Roughly speaking, this error measures how much the Bellman equality is violated.

where $Q(s, a)$ is now the output of the neural network.

There are several different architectural choices for using a neural network to approximate Q values:

- One network for each action a, that takes s as input and produces $Q(s, a)$ as output;
- One single network that takes s as input and produces a vector $Q(s, \cdot)$, consisting of the Q values for each action; or
- One single network that takes s, a concatenated into a vector (if a is discrete, we would probably use a one-hot encoding, unless it had some useful internal structure) and produces $Q(s, a)$ as output.

The first two choices are only suitable for discrete (and not too big) action sets. The last choice can be applied for continuous actions, but then it is difficult to find $\arg\max_{\mathcal{A}} Q(s, a)$.

For continuous action spaces, it is popular to use a class of methods called *actor-critic* methods, which combine policy and value-function learning. We won't get into them in detail here, though.

There are not many theoretical guarantees about Q-learning with function approximation and, indeed, it can sometimes be fairly unstable (learning to perform well for a while, and then getting suddenly worse, for example). But neural network Q-learning has also had some significant successes.

One form of instability that we do know how to guard against is *catastrophic forgetting*. In standard supervised learning, we expect that the training x values were drawn independently from some distribution. But when a learning agent, such as a robot, is moving through an environment, the sequence of states it encounters will be temporally correlated. For example, the robot might spend 12 hours in a dark environment and then 12 in a light one. This can mean that while it is in the dark, the neural-network weight-updates will make the Q function "forget" the value function for when it's light.

And, in fact, we routinely shuffle their order in the data file, anyway.

One way to handle this is to use *experience replay*, where we save our (s, a, s', r) experiences in a *replay buffer*. Whenever we take a step in the world, we add the (s, a, s', r) to the replay buffer and use it to do a Q-learning update. Then we also randomly select some number of tuples from the replay buffer, and do Q-learning updates based on them, as well. In general it may help to keep a *sliding window* of just the 1000 most recent experiences in the replay buffer. (A larger buffer will be necessary for situations when the optimal policy might visit a large part of the state space, but we like to keep the buffer size small for memory reasons and also so that we don't focus on parts of the state space that are irrelevant for the optimal policy.) The idea is that it will help you propagate reward values through your state space more efficiently if you do these updates. You can see it as doing something like value iteration, but using samples of experience rather than a known model.

11.2.3 Fitted Q-learning

An alternative strategy for learning the Q function that is somewhat more robust than the standard Q-learning algorithm is a method called *fitted Q*.

FITTED-Q-LEARNING($\mathcal{A}, s_0, \gamma, \alpha, \epsilon, m$)

```
1  s = s_0 // (e.g., s_0 can be drawn randomly from S)
2  D = { }
3  initialize neural-network representation of Q
4  while True:
5      D_new = experience from executing ε-greedy policy based on Q for m steps
6      D = D ∪ D_new represented as (s, a, s', r) tuples
7      D_supervised = {(x^(i), y^(i))} where x^(i) = (s, a) and y^(i) = r + γ max_{a'∈A} Q(s', a')
8        for each tuple (s, a, s', r)^(i) ∈ D
9      re-initialize neural-network representation of Q
10     Q = supervised_NN_regression(D_supervised)
```

Here, we alternate between using the policy induced by the current Q function to gather a batch of data $\mathcal{D}_{\text{new}}$, adding it to our overall data set $\mathcal{D}$, and then using supervised neural-network training to learn a representation of the Q value function on the whole data set. This method does not mix the dynamic-programming phase (computing new Q values based on old ones) with the function approximation phase (supervised training of the neural network) and avoids catastrophic forgetting. The regression training in line 9 typically uses squared error as a loss function and would be trained until the fit is good (possibly measured on held-out data).

11.2.4 Policy search

A different model-free strategy is to search directly for a good policy. The strategy here is to define a functional form $f(s;\theta) = a$ for the policy, where θ represents the parameters we learn from experience. We choose f to be differentiable, and often define $f(s, a; \theta) = \Pr(a|s)$, a conditional probability distribution over our possible actions.

Here, the conditional distribution simply means that the chance of choosing an action depends on which state the agent is in. For example, suppose a robot is trying to get to a goal and can go left or right. An unconditional policy could say: regardless of my state I go left 99 percent of the time; a conditional policy would take into consideration the robot's state, and say: if I'm to the right of the goal, I go left 99 percent of the time.

Now, we can train the policy parameters using gradient descent:

- When θ has relatively low dimension, we can compute a numeric estimate of the gradient by running the policy multiple times for different values of θ, and computing the resulting rewards.
- When θ has higher dimensions (e.g., it represents the set of parameters in a complicated neural network), there are more clever algorithms, e.g., one called REINFORCE, but they can often be difficult to get to work reliably.

Policy search is a good choice when the policy has a simple known form, but the MDP would be much more complicated to estimate.

11.3 Model-based RL

The conceptually simplest approach to RL is to model R and T from the data we have gotten so far, and then use those models, together with an algorithm for solving MDPs (such as value iteration) to find a policy that is near-optimal given the current models.

Assume that we have had some set of interactions with the environment, which can be characterized as a set of tuples of the form $(s^{(t)}, a^{(t)}, s^{(t+1)}, r^{(t)})$.

Because the transition function $T(s, a, s')$ specifies probabilities, multiple observations of (s, a, s') may be needed to model the transition function. One approach to this task of building a model $\hat{T}(s, a, s')$ for the true $T(s, a, s')$ is to estimate it using a simple counting strategy,

$$\hat{T}(s, a, s') = \frac{\#(s, a, s') + 1}{\#(s, a) + |\mathcal{S}|}. \tag{11.13}$$

Here, $\#(s, a, s')$ represents the number of times in our data set we have the situation where $s^{(t)} = s, a^{(t)} = a, s^{(t+1)} = s'$ and $\#(s, a)$ represents the number of times in our data set we have the situation where $s^{(t)} = s, a^{(t)} = a$.

Study Question: Prove to yourself that $\#(s, a) = \sum_{s'} \#(s, a, s')$.

Adding 1 and $|\mathcal{S}|$ to the numerator and denominator, respectively, are a form of smoothing called the *Laplace correction*. It ensures that we never estimate that a probability is 0, and keeps us from dividing by 0. As the amount of data we gather increases, the influence of this correction fades away.

In contrast, the reward function $R(s, a)$ (as we have specified it in this text) is a *deterministic* function, such that knowing the reward r for a given (s, a) is sufficient to fully determine the function at that point. In other words, our model $\hat{R}$ can simply be a record of observed rewards, such that $\hat{R}(s, a) = r = R(s, a)$.

Given empirical models $\hat{T}$ and $\hat{R}$ for the transition and reward functions, we can now solve the MDP $(\mathcal{S}, \mathcal{A}, \hat{T}, \hat{R})$ to find an optimal policy using value iteration, or use a search algorithm to find an action to take for a particular state.

This approach is effective for problems with small state and action spaces, where it is not too hard to get enough experience to model T and R well; but it is difficult to generalize this method to handle continuous (or very large discrete) state spaces, and is a topic of current research.

11.4 Bandit problems

Bandit problems differ from our reinforcement learning setting as described above in two ways: the reward function is probabilistic, and the key decision is usually framed as whether or not to continue exploring (to improve the model) versus exploiting (take actions to maximize expected rewards based on the current model).

A basic bandit problem is given by

- A set of actions $\mathcal{A}$;
- A set of reward values $\mathcal{R}$; and
- A probabilistic reward function $R_p : \mathcal{A} \times \mathcal{R} \to \mathbb{R}$, i.e., R_p is a function that takes an action and a reward and returns the probability of getting that reward conditioned on that action being taken, $R_p(a, r) = Pr(\text{reward} = r \,|\, \text{action} = a)$. This is analogous to how the transition function T is defined. Each time the agent takes an action, a new value is drawn from this distribution.

The most typical bandit problem has $\mathcal{R} = \{0, 1\}$ and $|\mathcal{A}| = k$. This is called a k-*armed bandit problem*, where the decision is which "arm" (action a) to select, and the reward is either getting a payoff (1) or not (0). There is a lot of mathematical literature on optimal strategies for k-armed bandit problems under various assumptions. The important question is usually one of *exploration versus exploitation*. Imagine that you have tried each action 10 times, and now you have estimates $\hat{R}_p(a, r)$ for the probabilities $R_p(a, r)$ for reward r given action a. Which arm should you pick next? You could

Why? Because in English slang, "one-armed bandit" is a name for a slot machine (an old-style gambling machine where you put a coin into a slot and then pull its arm to see if you get a payoff) because it has one arm and takes your money! What we have here is a similar sort of machine, but with k arms.

exploit your knowledge, and choose the arm with the highest value of expected reward on all future trials; or

explore further, by trying some or all actions more times, hoping to get better estimates of the $R_p(a, r)$ values.

The theory ultimately tells us that, the longer our horizon h (or, similarly, closer to 1 our discount factor), the more time we should spend exploring, so that we don't converge prematurely on a bad choice of action.

Study Question: Why is it that "bad" luck during exploration is more dangerous than "good" luck? Imagine that there is an action that generates reward value 1 with probability 0.9, but the first three times you try it, it generates value 0. How might that cause difficulty? Why is this more dangerous than the situation when an action that generates reward value 1 with probability 0.1 actually generates reward 1 on the first three tries?

Bandit problems are reinforcement learning problems (and are very different from batch supervised learning) in that:

- The agent gets to influence what data it obtains (selecting a gives it another sample from $R(a, r)$), and
- The agent is penalized for mistakes it makes while it is learning (if it is trying to maximize the expected reward $\sum_r r \cdot Pr(R_p(a, r) = r)$ it gets while behaving).

There is a setting of supervised learning, called *active learning*, where instead of being given a training set, the learner gets to select a value of x and the environment gives back a label y; the problem of picking good x values to query is interesting, but the problem of deriving a hypothesis from (x, y) pairs is the same as the supervised problem we have been studying.

In a *contextual* bandit problem, you have multiple possible states, drawn from some set S, and a separate bandit problem associated with each one.

Bandit problems are an essential subset of reinforcement learning. It's important to be aware of the issues, but we will not study solutions to them in this class.