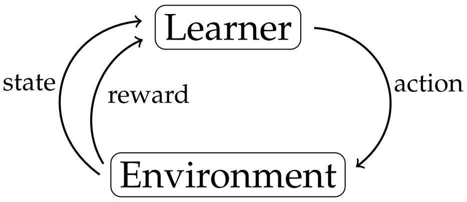
CHAPTER 11

Reinforcement learning

So far, all the learning problems we have looked at have been *supervised*: that is, for each training input $x^{(i)}$, we are told which value $y^{(i)}$ should be the output. A very different problem setting is *reinforcement learning*, in which the learning system is not directly told which outputs go with which inputs. Instead, there is an interaction of the form:

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

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



This problem setting is equivalent to an *online* supervised learning under the following assumptions:

1. The space of possible outputs is binary (e.g. $\{+1, -1\}$) and the space of possible rewards is binary (e.g. $\{+1, -1\}$);
2. $s^{(i)}$ is independent of all previous $s^{(j)}$ and $a^{(j)}$; and
3. $r^{(i)}$ depends only on $s^{(i)}$ and $a^{(i)}$.

In this case, for any experience tuple $(s^{(i)}, a^{(i)}, r^{(i)})$, we can generate a supervised training example, which is equal to $(s^{(i)}, a^{(i)})$ if $r^{(i)} = +1$ and $(s^{(i)}, -a^{(i)})$ otherwise.

Study Question: What supervised-learning loss function would this objective correspond to?

Reinforcement learning is more interesting when these properties do not hold. When we relax assumption 1 above, we have the class of *bandit problems*, which we will discuss in section 1. If we relax assumption 2, but assume that the environment that the agent is interacting with is an MDP, so that $s^{(i)}$ depends only on $s^{(i-1)}$ and $a^{(i-1)}$ then we are in the classical *reinforcement-learning* setting, which we discuss in section 2. Weakening the assumptions further, for instance, not allowing the learner to observe the current state completely and correctly, makes the problem into a *partially observed MDP* (POMDP), which is substantially more difficult, and beyond the scope of this class.

1 Bandit problems

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 $R : \mathcal{A} \times \mathcal{R} \rightarrow \mathbb{R}$, i.e. R is a function that takes an action and a reward and returns the probability of getting that reward conditioned on that action being taken, $R(a, r) = p(\text{reward} = r \mid \text{action} = a)$. 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*. 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 $\hat{R}(a_j, r)$ for the probabilities $R(a_j, r)$ for reward r given action a_j. 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 $\hat{R}(a_j, r)$ on all future trials; or

explore further, by trying some or all actions more times, hoping to get better estimates of the $R(a_j, r)$ 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?

Note that what makes this a very different kind of problem from the batch supervised learning setting is that:

- The agent gets to influence what data it gets (selecting a_j gives it another sample from $R(a_j, r)$), and
- The agent is penalized for mistakes it makes while it is learning (if it is trying to maximize the expected reward $\sum_r rR(a_j, r)$ 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 values 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 $\mathcal{S}$, and a separate bandit problem associated with each one.

Bandit problems will be an essential sub-component of reinforcement learning.

2 Sequential problems

In the more typical (and difficult!) case, we can think of our learning agent interacting with an MDP, where it knows $\mathcal{S}$ and $\mathcal{A}$, but not $T(s, a, s')$ or $R(s, a)$. The learner can interact with the environment by selecting actions. So, this is somewhat like a contextual bandit problem, but more complicated, because selecting an action influences not only what the immediate reward will be, but also what state the system ends up in at the next time step and, therefore, what additional rewards might be available in the future. Note that for simplicity, in the following we will use the deterministic reward function $R(s, a)$ defined in the last chapter, rather than the probabilistic one defined above for the bandit problem.

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_t 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. In the following sections, we will consider several different approaches.

2.1 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 $(s^{(t)}, a^{(t)}, r^{(t)}, s^{(t+1)})$.

Because the transition function $T(s, a, s')$ specifies probabilities, multiple observations of (s, a, s') may be needed to model the 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,

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

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 $\hat{R}(s,a) = r = R(s,a)$.

A more general case is when the reward function is *non-deterministic*, such that the reward $R(s,a)$ may vary with each query. In this case, an estimation strategy similar to that employed for the model $\hat{T}$ could be used for building the model $\hat{R}$, e.g.:

$$\hat{R}(s,a) = \frac{\sum r \mid s,a}{\#(s,a)}$$

where

$$\sum r \mid s,a = \sum_{\{t \mid s_t = s, a_t = a\}} r^{(t)} \quad .$$

Such an estimate would just be the average of the observed rewards for each s, a pair. However, this general case is outside the scope of what we will cover here.

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 finite-depth expecti-max search to find an action to take for a particular state.

This technique 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.

2.2 Policy search

A very different strategy is to search directly for a good policy, without first (or ever!) estimating the transition and reward models. 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 let $f(s;\theta) = P(a)$, a probability distribution over our possible actions.

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 $\theta \pm \epsilon$, and computing the resulting rewards.

- When θ has higher dimensions (e.g., it is 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 model would be much more complicated to estimate.

2.3 Value function learning

The most popular class of algorithms learns neither explicit transition and reward models nor a direct policy, but instead concentrates on learning a value function. It is a topic of current research to describe exactly under what circumstances value-function-based approaches are best, and there are a growing number of methods that combine value functions, transition and reward models and policies into a complex learning algorithm in an attempt to combine the strengths of each approach.

We will study two variations on value-function learning, both of which estimate the Q function.

2.3.1 Q-learning

This is the most typical way of performing reinforcement learning. Recall the value-iteration update:

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

We will adapt this update to the RL scenario, where we do not know the transition function T or reward function R.

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!

```
Q-LEARNING($\mathcal{S}, \mathcal{A}, s_0, \gamma, \alpha$)
for $s \in \mathcal{S}, a \in \mathcal{A}$ :
    $Q[s, a] = 0$
$s = s_0$ // Or draw an s randomly from $\mathcal{S}$
while True:
    $a = \text{select\_action}(s, Q)$
    $r, s' = \text{execute}(a)$
    $Q[s, a] = (1 - \alpha)Q[s, a] + \alpha(r + \gamma \max_{a'} Q[s', a'])$
    $s = s'$
```

Here, α represents the "learning rate," which needs to decay for convergence purposes, but in practice is often set to a constant.

Note that the update can be rewritten as

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

which looks something like a gradient update! This is often called *temporal difference* learning method, because we make an update based on the difference between the current estimated value of taking action a in state s, which is $Q[s, a]$, and the "one-step" sampled value of taking a in s, which is $r + \gamma \max_{a'} Q[s', a']$.

It is actually not a gradient update, but later, when we consider function approximation, we will treat it as if it were.

You can see this method as a combination of two different iterative processes that we have already seen: the combination of an old estimate with a new sample using a running average with a learning rate α, and the dynamic-programming update of a Q value from value iteration.

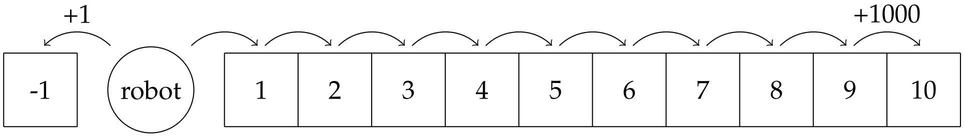
Our algorithm above includes a procedure called *select_action*, which, given the current state s, has to decide which action to take. If the Q value is estimated very accurately and the agent is behaving in the world, then generally we would want to choose the apparently optimal action $\arg\max_{a \in \mathcal{A}} Q(s, a)$. 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 stupid one).

A typical action-selection strategy is the 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

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 sample-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 for the last state on the hallway to have a high value, but it won't yet understand that moving to the right 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 $Q(0 : 10, \text{right})$, representing the Q values for moving right at each of the positions $0, \ldots, 9$. Then, for $\alpha = 1$ and $\gamma = 0.9$,

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

Starting with Q values of 0,

$$Q^{(0)}(0:10, \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

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

$$Q^{(1)}(0:10, \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1000 & 0 \end{bmatrix}$$

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:

$$Q^{(2)}(0:10, \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 900 & 1000 & 0 \end{bmatrix}$$

Similarly,

$$\begin{aligned} Q^{(3)}(0:10, \text{ right}) &= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 810 & 900 & 1000 & 0 \end{bmatrix} \\ Q^{(4)}(0:10, \text{ right}) &= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 729 & 810 & 900 & 1000 & 0 \end{bmatrix} \\ &\vdots \\ Q^{(10)}(0:10, \text{ right}) &= \begin{bmatrix} 387.4 & 420.5 & 478.3 & 531.4 & 590.5 & 656.1 & 729 & 810 & 900 & 1000 & 0 \end{bmatrix}, \end{aligned}$$

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

We can see how this interacts with the exploration/exploitation dilemma: from the perspective of s_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 the long hallway sufficiently.

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

2.3.2 Function approximation

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 the squared Bellman error, with loss:

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

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

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

- One network for each action a_j, that takes s as input and produces $Q(s, a_j)$ 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 increasingly 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 it 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. 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.

For example, it might spend 12 hours in a dark environment and then 12 in a light one.

One way to handle this is to use *experience replay*, where we save our (s, a, r, s') experiences in a *replay buffer*. Whenever we take a step in the world, we add the (s, a, r, s') 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.

2.3.3 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  $s = s_0$ // Or draw an $s$ randomly from $\mathcal{S}$
2  $\mathcal{D} = \{\,\}$
3  initialize neural-network representation of Q
4  while True:
5      $\mathcal{D}_{\text{new}}$ = experience from executing $\epsilon$-greedy policy based on Q for $m$ steps
6      $\mathcal{D} = \mathcal{D} \cup \mathcal{D}_{\text{new}}$ represented as $(s, a, r, s')$ tuples
7      $D_{\text{sup}} = \{(x^{(i)}, y^{(i)})\}$ where $x^{(i)} = (s, a)$ and $y^{(i)} = r + \gamma \max_{a' \in \mathcal{A}} Q(s', a')$
8        for each tuple $(s, a, r, s')^{(i)} \in \mathcal{D}$
9      re-initialize neural-network representation of Q
10     $Q = \text{supervised\_NN\_regression}(D_{\text{sup}})$
```

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 (training 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).