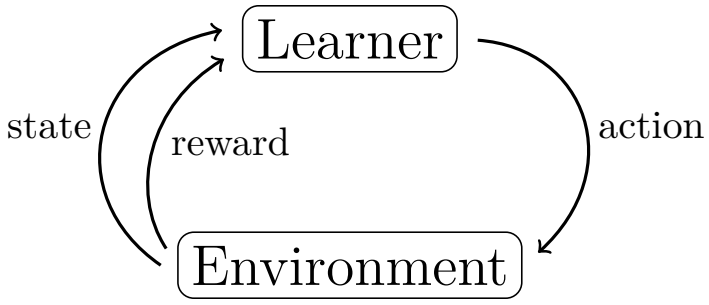
11 Reinforcement Learning

> **Note**
>
> This page contains all content from the legacy PDF notes; reinforcement learning chapter.
>
> As we phase out the PDF, this page may receive updates not reflected in the static PDF.

Reinforcement learning (RL) is a type of machine learning where an agent learns to make decisions by interacting with an environment. Unlike other learning paradigms, RL has several distinctive characteristics:

- The agent interacts directly with an environment, receiving feedback in the form of rewards or penalties
- The agent can choose actions that influences what information it gains from the environment
- The agent updates its decision-making strategy incrementally as it gains more experience

In a reinforcement learning problem, the interaction between the agent and environment follows a specific pattern:



The interaction cycle proceeds as follows:

1. Agent observes the current *state* $s^{(i)}$
2. Agent selects and executes an *action* $a^{(i)}$
3. Agent receives a *reward* $r^{(i)}$ from the environment
4. Agent observes the new *state* $s^{(i+1)}$
5. Agent selects and executes a new *action* $a^{(i+1)}$
6. Agent receives a new *reward* $r^{(i+1)}$
7. This cycle continues…

Similar to MDP Chapter 10, in an RL problem, the agent's goal is to learn a *policy* - a mapping from states to actions - that maximizes its expected cumulative reward over time. This policy guides the agent's decision-making process, helping it choose actions that lead to the most favorable outcomes.

11.1 Reinforcement learning algorithms overview

Approaches to reinforcement learning differ significantly according to what kind of hypothesis or model is being learned. Roughly speaking, RL methods can be categorized into model-free methods and model-based methods. The main distinction is that model-based methods explicitly learn the transition and reward models to assist the end-goal of learning a policy; model-free methods do not. We will start our discussion with the model-free methods, and introduce two of the arguably most popular types of algorithms, Q-learning Section 11.1.2 and policy gradient Section 11.3. We then describe model-based methods Section 11.4. Finally, we briefly consider "bandit" problems Section 11.5, which differ from our MDP learning context by having probabilistic rewards.

11.1.1 Model-free methods

Model-free methods are methods that do not explicitly learn transition and reward models. Depending on what is explicitly being learned, model-free methods are sometimes further categorized into value-based methods (where the goal is to learn/estimate a value function) and policy-based methods (where the goal is to directly learn an optimal policy). 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.

11.1.2 Q-learning

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

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

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 instead rely on samples to perform the updates.

---

procedure Q-Learning($\mathcal{S}, \mathcal{A}, \gamma, \alpha, s_0, \text{max\_iter}$)
  $i \leftarrow 0$
  for all $s \in \mathcal{S},\ a \in \mathcal{A}$ **do**
    $Q_{\text{old}}(s,a) \leftarrow 0$
  end for
  $s \leftarrow s_0$
  while $i <$ max_iter **do**

```
    a ← select_action(s, Q_old(s, a))
    (r, s') ← execute(a)
    Q_new(s, a) ← (1 − α) Q_old(s, a) + α(r + γ max_{a'} Q_old(s', a'))
    s ← s'
    i ← i + 1
    Q_old ← Q_new
  end while
  return Q_new
end procedure
```

With the pseudo-code provided for Q-Learning, there are a few key things to note. First, we must determine which state to initialize the learning from. In the context of a game, this initial state may be well defined. In the context of a robot navigating an environment, one may consider sampling the initial state at random. In any case, the initial state is necessary to determine the trajectory the agent will experience as it navigates the environment.

Second, different contexts will influence how we want to choose when to stop iterating through the while loop. Again, in some games there may be a clear terminating state based on the rules of how it is played. On the other hand, a robot may be allowed to explore an environment *ad infinitum*. In such a case, one may consider either setting a fixed number of transitions (as done explicitly in the pseudo-code) to take; or we may want to stop iterating in the example once the values in the Q-table are not changing, after the algorithm has been running for a while.

Finally, a single trajectory through the environment may not be sufficient to adequately explore all state-action pairs. In these instances, it becomes necessary to run through a number of iterations of the Q-Learning algorithm, potentially with different choices of initial state s_0.

This notion of running a number of instances of Q-Learning is often referred to as experiencing multiple *episodes*.

Of course, we would then want to modify Q-Learning such that the Q table is not reset with each call.

Now, let's dig into what is happening in Q-Learning. Here, $\alpha \in (0, 1]$ 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 specifically as tabular Q-learning.

In the Q-learning update rule

$$Q[s, a] \leftarrow (1 - \alpha)Q[s, a] + \alpha(r + \gamma \max_{a'} Q[s', a']) \tag{11.1}$$

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.

Equation 11.1 can also be equivalently rewritten as

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

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 and current Q function, has to decide which action to take. If the Q value is estimated very accurately and the agent is deployed to "behave" in the world (as opposed to "learn" 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).

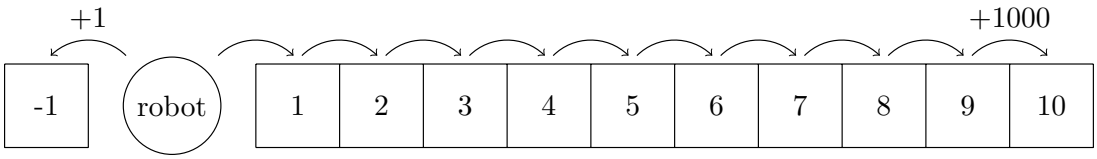
A typical action-selection strategy that attempts to address this *exploration versus exploitation* dilemma is the so-called ϵ-*greedy* strategy:

- 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! The conditions specified in the theorem are: visit every state-action pair infinitely often, and the learning rate α satisfies a scheduling condition. This implies that for exploration strategy specifically, any strategy is okay as long as it tries every state-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 $\gamma = 0.9$, Equation 11.2 becomes

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

Starting with Q values of 0,

$$Q^{(0)}(i = 0, \ldots, 9;\ \text{right}) = [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]\,.$$

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.

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

$$Q^{(1)}(i = 0, \ldots, 9;\ \text{right}) = [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1000]\,.$$

After making its way down the hallway again,

$$Q(8, \text{right}) = 0 + 0.9\, Q(9, \text{right}) = 900$$

updates:

$$Q^{(2)}(i = 0, \ldots, 9;\ \text{right}) = [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 900 \quad 1000]\,.$$

Similarly,

$$Q^{(3)}(i = 0, \ldots, 9;\ \text{right}) = [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 810 \quad 900 \quad 1000]\,,$$

$$Q^{(4)}(i = 0, \ldots, 9;\ \text{right}) = [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 729 \quad 810 \quad 900 \quad 1000]\,,$$

...

$$Q^{(10)}(i = 0, \ldots, 9; \text{right}) = [387.4 \quad 420.5 \quad 478.3 \quad 531.4 \quad 590.5 \quad 656.1 \quad 729 \quad 810 \quad 900 \quad 1000],$$

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

Here, we can see the exploration/exploitation dilemma in action: from the perspective of $s_0 = 0$, it will seem that getting the immediate reward of 1 is a better strategy without exploring the long hallway.

> **? Study Question**
>
> Determine the Q value functions that result from always executing the "move left" policy.

11.2 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 inputs s and a, and outputs $Q(s, a)$. We can treat this as a regression problem, optimizing this loss:

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

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 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.

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.

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 $\arg\max_{a \in \mathcal{A}} Q(s, a)$.

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 suddenly getting 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.

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

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.

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 us propagate reward values through our state space more efficiently if we do these updates. We can see it as doing something like value iteration, but using samples of experience rather than a known model.

11.2.1 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*.

procedure FITTED-Q-LEARNING($\mathcal{A}, s_0, \gamma, \alpha, \epsilon, m$)
  $s \leftarrow s_0$//e.g., s_0 can be drawn randomly from $\mathcal{S}$
  $\mathcal{D} \leftarrow \emptyset$
  initialize neural-network representation of Q
  while True **do**
    $\mathcal{D}_{\text{new}} \leftarrow$ experience from executing ϵ-greedy policy based on Q for m steps
    $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_{\text{new}}$ represented as tuples (s, a, s', r)
    $\mathcal{D}_{\text{supervised}} \leftarrow \emptyset$
    for each tuple $(s, a, s', r) \in \mathcal{D}$ **do**
      $x \leftarrow (s, a)$
      $y \leftarrow r + \gamma \max_{a' \in \mathcal{A}} Q(s', a')$
      $\mathcal{D}_{\text{supervised}} \leftarrow \mathcal{D}_{\text{supervised}} \cup \{(x, y)\}$
    end for
    re-initialize neural-network representation of Q
    $Q \leftarrow$ supervised-NN-regression($\mathcal{D}_{\text{supervised}}$)
  end while
end procedure

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 10 typically uses squared error as a loss function and would be trained until the fit is good (possibly measured on held-out data).

11.3 Policy gradient

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

This means the chance of choosing an action depends on which state the agent is in. Suppose, e.g., a robot is trying to get to a goal and can go left or right. An unconditional policy can say: I go left 99% of the time; a conditional policy can consider the robot's state, and say: if I'm to the right of the goal, I go left 99% of the time.

$f(s, a; \theta) = Pr(a|s)$, a conditional 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 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.

11.4 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 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 $s^{(t)} = s$, $a^{(t)} = a$.

Conceptually, this is similar to having "initialized" our estimate for the transition function with uniform random probabilities before making any observations.

Adding 1 and $|\mathcal{S}|$ to the numerator and denominator, respectively, is 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 $R(s, a)$ 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. 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.

11.5 Bandit problems

Bandit problems are a subset of reinforcement learning 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 $R_p : \mathcal{A} \times \mathcal{R} \to \mathbb{R}$, 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,
  $R_p(a, r) = Pr(\text{reward} = r \mid \text{action} = a)$. Each time the agent takes an action, a new value is drawn from this distribution.

Notice that this probablistic rewards set up in bandits differs from the "rewards are deterministic" assumptions we made so far.

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).

Why "bandit"? In English slang, "one-armed bandit" refers to a slot machine because it has one arm and takes your money! Here, we have a similar machine but with k arms.

The important question is usually one of *exploration versus exploitation*. Imagine you have tried each action 10 times, and now you have estimates $\hat{R}_p(a, r)$ for the probabilities $R_p(a, r)$. Which arm should you pick next? You could:

- *exploit* your knowledge, choosing the arm with the highest value of expected reward; or

- *explore* further, trying some or all actions more times 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.

Bandit problems are reinforcement learning problems (and 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 (trying to maximize the expected reward it gets while behaving).

In a *contextual bandit problem*, you have multiple possible states from some set $\mathcal{S}$, 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.