Temporal Difference Learning and Q-Learning

In the previous chapter, we covered several reinforcement learning algorithms including Fitted Q-Iteration and Approximate Policy Iteration. These methods are sometimes called *batch methods* or *offline methods* because a batch of samples a collected offline, and a fitted value function (or action-value function) is found by minimizing the training error for these samples. Offline methods like these make efficient use of available training data, but are computationally expensive and suffer from high memory consumption as the number of samples increases. In this lecture, we present several online techniques that perform an incremental update after each state transition (x, a, r, x'). Note that in this chapter we have switched from cost to reward r, as is common in the reinforcement learning literature. Hence online methods can learn a policy with relatively low computational and memory cost because the updates are made based on a single state transition. Sometimes people refer to the state transition (x, a, r, x') as an *experience*.

First, we will present the *Temporal-Difference (TD) method* for online policy evaluation. Next, we present another algorithm called *SARSA* that extends online policy evaluation to the action-value function. Finally, we explore *Q-learning* as a method for finding the action-value function for the optimal policy, and hence finding the optimal policy.

In this lecture, we consider the infinite time horizon case and assume a deterministic policy. Recall that the value function for a fixed policy π is defined as

$$V^{\pi}(x) = E\left[\sum_{t=0}^{\infty} \gamma^t \, r(x_t, \pi(x_t))\right], \quad \text{where } x_0 = x. \tag{0.0.43}$$

The action-value function for a fixed policy π is defined as

$$Q^{\pi}(x, a) = r(x, a) + E\left[\sum_{t=1}^{\infty} \gamma^t \, r(x_t, \pi(x_t))\right], \quad \text{where } x_0 = x. \tag{0.0.44}$$

The Bellman Equations in this case,

$$\begin{aligned} V^{\pi}(x) &= r(x, \pi(x)) + \gamma \, \mathbb{E}_{p(x'|x,\pi(x))}[V^*(x')] \\ Q^{\pi}(x, a) &= r(x, a) + \gamma \, \mathbb{E}_{p(x'|x,a)}[Q^{\pi}(x', \pi(x'))]. \end{aligned} \tag{0.0.45}$$

The Bellman Equations for the the value function V^* and action-value function Q^* of the optimal policy π^* are,

$$\begin{aligned} V^*(x) &= \max_{a' \in \mathbb{A}} \left(r(x, a) + \gamma \, \mathbb{E}_{p(x'|x,a)}[V^*(x')] \right) \\ Q^*(x, a) &= r(x, a) + \gamma \, \mathbb{E}_{p(x'|x,a)}[\max_{a' \in \mathbb{A}} Q^*(x', a')]. \end{aligned} \tag{0.0.46}$$

Temporal-Difference Learning

Temporal-difference (TD) Learning, is an online method for estimating the value function for a fixed policy π. The main idea behind TD-learning is that we can learn about the value function from *every* experience (x, a, r, x') as a robot traverses the environment.

Given an estimate of the value function $\tilde{V}^{\pi}(x)$ we would like to perform an update in order to minimize the squared loss[5],

$$\mathcal{L} = \frac{1}{2} \left(V^{\pi}(x) - \tilde{V}^{\pi}(x) \right)^2. \tag{0.0.47}$$

[5] Technically, this squared loss is still an estimate of the Bellman error $E_{d^{\pi}(x)}[\frac{1}{2} \left(V^{\pi}(x) - \tilde{V}^{\pi}(x) \right)^2]$, where $d^{\pi}(x)$ is the probability of a state x being visit under policy π.

Since we do not yet know the value function, evaluating this loss requires evaluating equation (0.0.43). Naïvely, this method would require waiting until the end of an episode before updating $\tilde{V}^{\pi}(x)$. Instead, we estimate $V^{\pi}(x)$ as $y = r + \gamma \, \tilde{V}^{\pi}(x')$ and perform an online update for each experience $(x, \pi(x), r, x')$. Plugging this estimate into the loss function we get

$$\mathcal{L}_{\text{approx}} = \frac{1}{2} \left(y - \tilde{V}^{\pi}(x) \right)^2. \tag{0.0.48}$$

The gradient of eq. (0.0.48) with respect to $\tilde{V}^{\pi}$ is:

$$\nabla_{\tilde{V}^{\pi}(x)} \mathcal{L}_{\text{approx}} = \left(y - \tilde{V}^{\pi}(x) \right) \left(\nabla_{\tilde{V}^{\pi}(x)} y - 1 \right). \tag{0.0.49}$$

If our state-space is discrete and $V^{\pi}(x')$ and $V^{\pi}(x)$ are independent, $\nabla_{\tilde{V}^{\pi}(x)} y = 0$.

$$\begin{aligned} \tilde{V}^{\pi}(x) &\leftarrow \tilde{V}^{\pi}(x) + \alpha \left(r + \gamma \tilde{V}^{\pi}(x') - \tilde{V}^{\pi}(x) \right) \\ &\leftarrow (1 - \alpha) \tilde{V}^{\pi}(x) + \alpha \left(r + \gamma \tilde{V}^{\pi}(x') \right). \end{aligned} \tag{0.0.50}$$

The term $\left(r + \gamma \tilde{V}^{\pi}(x') - \tilde{V}^{\pi}(x) \right)$ is sometimes known as *TD error*.

By looking at the second line of (0.0.50), one may notice that TD-learning is also closely related to an *exponential moving average*.

Algorithm TD

The TD-learning algorithm is shown in Algorithm 15.

Initialize $\tilde{V}^{\pi}$
while $\tilde{V}^{\pi}$ *not converged* **do**
  Initialize x according to a particular starting state
  while x *is not a terminal state* **do**
    apply action $a \leftarrow \pi(x)$
    receive experience $(x, \pi(x), r, x')$
    update $\tilde{V}^{\pi}(s)$

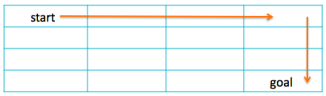
$$\tilde{V}^{\pi}(s) \leftarrow (1-\alpha)\tilde{V}^{\pi}(x) + \alpha\left(r + \gamma\tilde{V}^{\pi}(x')\right)$$

    set $x \leftarrow x'$
  end
end
return $\tilde{V}^{\pi}$

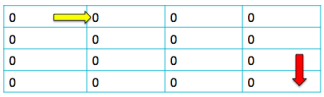
Algorithm 14: The TD-learning algorithm.

Grid-World Example

The diagram below shows a grid-based world, where the robot starts in the upper left $(0,0)$, and the goal is in the lower right $(3,3)$. The robot gets a reward of $+1$ if it reaches the goal, and 0 everywhere else. There is a discount factor of γ. The policy is for the robot to go right until it reaches the wall, and then go down.



We start by initializing $\tilde{V}^{\pi}(x) = 0,\ \forall x \in \mathbb{X}$.



As the robot moves one cell over from the start state (yellow arrow above), the reward is 0, and the value of both the current state and the next state is 0, so the approximate gradient used in the update rule (0.0.50) evaluates to 0 and no update is performed. As the robot moves into the goal state (red arrow), the reward is 1, so the approximate gradient evaluates to 1. We then update the second-to-last cell with (0.0.50) and we get:

$$\begin{aligned}\tilde{V}^{\pi}((3,2)) &\leftarrow (1-\alpha)\tilde{V}^{\pi}((3,2)) + \alpha\left(1+\gamma\tilde{V}^{\pi}((3,3))\right)\\ &= (1-\alpha)\times 0 + \alpha\times(1+0) = \alpha.\end{aligned}$$

0	0	0	0
0	0	0	0
0	0	0	α
0	0	0	0

Another iteration of the algorithm gives us:

$$\begin{aligned}\tilde{V}^{\pi}((3,2)) &\leftarrow (1-\alpha)\tilde{V}^{\pi}((3,2)) + \alpha\left(1+\gamma\tilde{V}^{\pi}((3,3))\right)\\ &= (1-\alpha)\times\alpha + \alpha\times(1+0)\\ &= \alpha + \alpha(1-\alpha),\\ \tilde{V}^{\pi}((3,1)) &\leftarrow (1-\alpha)\tilde{V}^{\pi}((3,1)) + \alpha\left(1+\gamma\tilde{V}^{\pi}((3,2))\right)\\ &= (1-\alpha)\times 0 + \alpha\times(0+\gamma\times\alpha)\\ &= \alpha^2\gamma.\end{aligned}$$

0	0	0	0
0	0	0	$\alpha^2\gamma$
0	0	0	α + α(1-α)
0	0	0	0

This method is slow, because we have to run the whole policy just to update the next cell. We will see that SARSA and Q-learning has similar issues of inefficient usage of experience.

SARSA

SARSA extends the Temporal-Difference method presented in the previous section to evaluate policies represented by a action-value functions $Q^{\pi}(x,a)$. Similar to the TD case, we wish to evaluate a policy by performing an online update to obtain an estimate, $\tilde{Q}^{\pi}(x,a)$, of the true action-value function $Q^{\pi}(x,a)$:

$$Q^{\pi}(x,a) = r(x,a) + \sum_{t=1}^{\infty}\gamma^t E[r(x_t,\pi(x_t))] \tag{0.0.51}$$

As in TD, we seek to minimize the loss

$$\mathcal{L}_{\text{approx}} = \frac{1}{2}\left(y - \tilde{Q}^{\pi}(x,a)\right)^2 \tag{0.0.52}$$

where $y = r(x,a) + \gamma\tilde{Q}^{\pi}(x',\pi(x'))$. Following a similar derivation as used for the TD update, we arrive at the SARSA update rule:

$$\tilde{Q}^{\pi}(x,a) \leftarrow (1-\alpha)\tilde{Q}^{\pi}(x,a) + \alpha\left[r(x,a) + \gamma\tilde{Q}^{\pi}(x',\pi(x'))\right]. \tag{0.0.53}$$

Algorithm SARSA

The SARSA algorithm is shown in Algorithm 16.

Initialize $\tilde{Q}^{\pi}$
while $\tilde{Q}^{\pi}$ *not converged* **do**
  Initialize x according to a particular starting state
  while x *is not a terminal state* **do**
    apply action $a \leftarrow \pi(x)$
    receive experience $(x, \pi(x), r, x', \pi(x'))$
    update $\tilde{Q}^{\pi}(s)$

$$\tilde{Q}^{\pi}(x,a) \leftarrow (1-\alpha)\tilde{Q}^{\pi}(x,a) + \alpha\left[r(x,a) + \gamma\tilde{Q}^{\pi}(x', \pi(x'))\right]$$

    set $x \leftarrow x'$
  end
end
return $\tilde{V}^{\pi}$

Algorithm 15: The TD-learning algorithm.

One may notice that TD-learning and SARSA are essentially approximate policy evaluation algorithms for the current policy. As a result of that they are examples of *on-policy* methods that can only use samples from the *current* policy to update the value and Q function. As we will see later, Q learning, on the contrary, is an *off-policy* method that can use samples from *any* policies to update the Q function.

Q-Learning

Q-Learning attempts to learn the *optimal* action-value function $Q^*(x,a)$ from an online stream of experiences. Recall that the Bellman Equation for the optmal action-value function $Q^*(x,a)$ is,

$$Q^*(x,a) = r(x,a) + \gamma\, \mathbb{E}_{p(x'|x,a)}[\max_{a' \in \mathbb{A}} Q^*(x',a')].$$

Suppose we receive experience (x,a,r,x'). If the transition model is deterministic, we would update the action-value function as,

$$\tilde{Q}^*(x,a) \leftarrow r + \gamma \max_{a' \in \mathbb{A}} \tilde{Q}^*(x',a').$$

However, just as in SARSA, this performs poorly when the transition or reward functions are stochastic. Instead, we update $\tilde{Q}^*$ to the weighted sum,

$$\tilde{Q}^*(x,a) \leftarrow \alpha\left[r + \gamma \max_{a' \in \mathbb{A}} \tilde{Q}^*(x',a')\right] + (1-\alpha)\tilde{Q}^*(x,a),$$

where $0 \leq \alpha \leq 1$ is the *learning rate*.

One may notice that we do not need the current policy π to update $\tilde{Q}^*$. Moreover, Q-learning approximates the *optimal* action-value function, the Bellman Equation of which does not depend on the specific policy that the agent is executing. Therefore, Q-learning is an *off-policy* algorithm that can use samples from *any* policies to update $\tilde{Q}^*$.

Q-learning is guaranteed to converge $\tilde{Q}^*$ to the optimal action-value function Q^* as number of iterations $k \to \infty$ given that the following conditions hold:

1. Each state-action pair is visited infinite times
2. $\lim_{k\to\infty} \sum_{k=0}^{\infty} \alpha_k = \infty$
3. $\lim_{k\to\infty} \sum_{k=0}^{\infty} \alpha_k^2 < \infty$,

where α_k is the learning rate at iteration k. The latter two conditions mean that the learning rate α must be annealed over time. Intuitively, this means that the agent begins by quickly updating $\tilde{Q}^*$, then slows down to refine its estimate as it receives more experience.

Fitted Q-Learning

Just as the fitted Q-iteration algorithm, we can use a function approximator to approximate the action-value function.

Suppose that we approximate Q^* with the function Q_θ with parameter θ. Instead of directly updating our action-value function, we now must update θ to achieve the desired change in Q_θ.

To fit θ, we minimize a loss function

$$\mathcal{L} = \frac{1}{2}\left(y - Q_\theta(x,a)\right)^2$$

that penalizes deviation between the approximate action-value function $Q_\theta(x,a)$ and the value $y = r + \gamma \max_{a' \in A} Q_\theta(x',a')$ predicted by a Bellman backup.

First, we must derive the gradient of $\mathcal{L}$. By applying the chain rule, we find

$$\begin{aligned}\nabla_\theta \mathcal{L} &= \left(y - Q_\theta(x,a)\right)\left[\nabla_\theta y - \nabla_\theta Q_\theta(x,a)\right] \\ &= \left(y - Q_\theta(x,a)\right)\left[\gamma \nabla_\theta Q_\theta(x',a^*) - \nabla_\theta Q_\theta(x,a)\right]\end{aligned}$$

where $a^* = \text{argmax}_{a' \in \mathbb{A}} Q_\theta(x',a')$ is the optimal action according to Q^θ. Unfortunately, it is not possible to obtain an unbiased estimate of $Q_\theta(x,a)\nabla_\theta Q_\theta(x',a^*)$ using one sample (x,a,r,x'). We can find the optimal parameter θ by performing gradient descent on $\mathcal{L}$ with the update rule,

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}. \tag{0.0.54}$$

Q-learning, however, assumes that y is constant and approximates the gradient as

$$\widetilde{\nabla}_\theta \mathcal{L} = -\left(y - Q_\theta(x,a)\right) \nabla_\theta Q_\theta(x,a). \tag{0.0.55}$$

The complete fitted Q-learning update rule is found by substituting eq. (0.0.55) into eq. (0.0.54):

$$\begin{aligned}\theta &\leftarrow \theta + \alpha \left[y - Q_\theta(s,a)\right] \nabla Q_\theta(x,a) \\ &\leftarrow \theta + \alpha \left[\left(r - \gamma Q_\theta(x', a^*)\right) - Q_\theta(x,a)\right] \nabla Q_\theta(x,a).\end{aligned}$$

Bellman Residual Method

Fitted Q-learning as described above does *not* implement gradient descent and, thus, is not guaranteed to converge to a local minimum. The *Bellman residual algorithm* avoids the approximation of eq. (0.0.55) by estimating the true gradient $\nabla_\theta \mathcal{L}$.

$$\nabla_\theta \mathcal{L} = \left(y - Q_\theta(x,a)\right)\left(\gamma \nabla_\theta Q_\theta(x', a^*) - \nabla_\theta Q_\theta(x,a)\right).$$

This estimation is only unbiased if we can generate two or more independent successor states for taking action a in state s. Generating these samples is trivial if we are able to simulate the system; i.e. have access to a known or learned transition model. If we do not know the transition model, then it is only possible to perform a Bellman residual update if we postpone a backup until the same state-action pair has been observed two or more times. This is often impossible when learning on a real system that has a continuous state-action space.

Exploration Policies

Unlike SARSA, which is an *on-policy* method, Q-learning is an *off-policy* method that can learn from arbitrary (x, a, r, x') experiences, regardless of what policy was used to generate them. This means that it is possible to use an *exploration policy* training that encourages the agent to visit previously unexplored regions of the state-action space. Exploration policies guarantee that the agent visits each state an infinite number of times and ensure convergence when the function is represented by a look-up table.

Two exploration policies that are commonly used with Q-learning are:

1. **ϵ-Greedy**. Choose the greedy action $a = \text{argmax}_{a \in \mathbb{A}} \tilde{Q}(x,a)$ with probability $1 - \epsilon$. Otherwise, with probability ϵ, choose an action uniformly at random $a \sim \text{uniform}(\mathbb{A})$. Higher values of ϵ encourage more exploration. Usually we set ϵ close to 1 as learning starts, and decay $\epsilon \to 0$ as we go along.

2. **Boltzmann Exploration.** Choose action a with probability

$$\pi(a|x) = \frac{\exp\left[\beta Q(x,a)\right]}{\sum_{a' \in \mathbb{A}} \exp\left[\beta Q(x,a')\right]},$$

which is weighted towards selecting actions with higher Q-values. Lower values of β encourage more exploration: the exploration policy with $\beta = 0$ is essentially a uniform distribution, as $\beta \to \infty$ the exploration policy becomes the greedy policy

$$\pi(a|x) = \arg\max_{a' \in \mathbb{A}} Q(x,a').$$

Hence, we usually start with β close to 0 and gradually increase β.

Experience Replay

Q-learning and SARSA are computationally efficient, but make inefficient use of data. Unlike batch methods, each sample is only used exactly once. This means that the agent must observe each transition ((x,a,r,x') for Q-learning and (x,a,r,x',a') for SARSA) many times to propagate the reward backwards in time.

Experience relay allows Q-learning to re-use experience multiple times by building a database D of experiences under the currently policy, sometimes called the *replay buffer*. Once enough data has been collected, the agent performs a fixed number of Q-learning or SARSA updates on the batch. This technique bridges the gap between offline methods and online methods, and can potentially combine the advantages the two.

Moreover, because Q-learning is an off-policy algorithm, the experiences generated from previous trajectories and policies can be *re-used* to update the estimate of action-value functions. Therefore, we can use a replay buffer across Q-learning updates: every time a new experience is generated, it is added to the replay buffer, and the agent performs Q-learning updates using *random samples* from the replay buffer.

Experience relay also helps address the problem of *correlated samples* for fitted Q-learning. In the case of online updates, the a experience is likely to be highly correlated with the previous/next experience because they are from the same trajectory. This makes the function approximator easily *overfit* to the current part of the state space, but fail to perform well for the entire state space. However, such correlation is mitigated when we use a batch of samples from possibly different trajectories to update the function approximator.