TD-learning

Catherine Ji

February 28, 2025

1 Bellman Equation Review

As a brief review of the Bellman equation (Q-function version), we can express the Q-function as

$$Q^\pi(s,a) = r(s,a) + \mathbb{E}_{s'\sim p(\cdot|s,a)}[\gamma \mathbb{E}_{a'\sim\pi(a'|s')}[Q^\pi(s',a')]] \tag{1}$$

$$= r(s,a) + \gamma \mathbb{E}_{s'\sim p(s'|s,a)}[V^\pi(s')]. \tag{2}$$

In tabular Q-learning, we assumed that we had access to the reward and transition functions. However, in the RL problem, we do not have access to these functions. So, what do we do?

2 When does Monte Carlo not work?

One naive approach to learn Q-functions and value functions is to use Monte Carlo methods – aka, we sample a bunch of trajectories starting from (s,a) and compute the average full-trajectory returns.

However, Monte Carlo methods are not always the best choice because we may need an exponentially-large (in horizon) number of trajectories to get coverage over our trajectory space. This makes Monte Carlo extremely sample inefficient...

3 Temporal Difference Learning

So, what can we do? Somehow, we want to leverage learned Q-functions at different states to help us learn the Q-function at a new state – after all, we transition between states to obtain the total reward!

And how do we relate the value functions at different states? (Ask students.) The answer is the Bellman Equation! **Temporal Difference Learning** converts the Bellman Equation into an actual algorithm when we are trying to solve the RL problem. Namely, **we do not have access to the reward and transition functions** (unlike the setting in your last homework assignment).

Definition 1 (Temporal Difference Learning)

Temporal Difference (TD) Learning is a model-free RL algorithm that, as the name suggests, learns from the difference between two estimates of the value functions computed at the "current" and "next" time steps. Namely, the difference between:

$$\begin{aligned} \text{current estimate} &= \hat{Q}^\pi(s,a) \\ \text{next estimate} &= r(s,a) + \gamma \hat{Q}^\pi(s',a') \\ \text{TD-error} = \text{current estimate} - \text{next estimate} &= \hat{Q}^\pi(s,a) - \left(r(s,a) + \gamma \hat{Q}^\pi(s',a')\right) \end{aligned}$$

should signal to us how to update our current estimate – we want to minimize this TD-error! Formally, a TD method makes the update:

$$\hat{Q}^\pi(s,a) \leftarrow (1-\alpha)\hat{Q}^\pi(s,a) + \alpha\left(r(s,a) + \gamma \hat{Q}^\pi(s',a')\right)$$

where α is the learning rate.

3.1 SARSA Algorithm

The SARSA algorithm is a TD method that is on-policy, meaning that it learns the Q-function for the policy that it is currently following. The algorithm is as follows:

Algorithm 1 SARSA Algorithm

1: **Given:** a set of transitions $\mathcal{T} = \{(s_0, a_0, r_0, s_0', a_0'), \dots\}$ collected from policy π
2: Initialize $Q^\pi(s,a)$ arbitrarily
3: **for** each $(s_i, a_i, r_i, s_i', a_i') \in \mathcal{T}$ **do**
4: $\quad \hat{Q}^\pi(s_i,a_i) \leftarrow (1-\alpha)\hat{Q}^\pi(s_i,a_i) + \alpha(r + \gamma \hat{Q}^\pi(s_i',a_i'))$
5: **end for**
6: **Update:** $Q^\pi(s,a) \leftarrow \hat{Q}^\pi(s,a)$

The algorithm is named SARSA, because you have access to (s,a,r,s',a') and learn the value function. Note that the update rule is a Monte Carlo estimate of the Bellman Update. The key difference between Value Iteration and SARSA is that SARSA is a model-free algorithm: recall that we needed access to transitions and rewards in Value Iteration!

3.2 Expected SARSA

Expected SARSA is a slight modification of the SARSA algorithm. Instead of sampling the next action a' given by the policy that gives our data, we can take the next action given an arbitrary policy. This is a bit magical – we are getting the Value Function for an arbitrary policy with access to data that comes from a different policy! We call this an **off-policy** method – we do not need to collect trajectories with the policy we're trying to evaluate.

The algorithm, given a set of trajectories $\mathcal{T}$ (which may follow a policy different from π) and a policy π, is as follows:

Algorithm 2 Expected SARSA Algorithm

Given: a set of trajectories $\mathcal{T} = \{(s_0, a_0, r_0, s_0'), \dots\}$ from π and a (possibly different) policy π'
Initialize $Q^{\pi'}(s, a)$ arbitrarily
for each $(s_i, a_i, r_i, s_i') \in \mathcal{T}$ **do**
    $\hat{Q}^{\pi'}(s_i, a_i) \leftarrow (1 - \alpha)\hat{Q}^{\pi'}(s_i, a_i) + \alpha[r + \gamma \hat{Q}^{\pi'}(s_i', a' \sim \pi'(\cdot \mid s_i'))]$
end for

4 Converting TD-learning methods to an actual optimal policy

But, in general, we don't just want the Q-function for policies – we want the optimal Q-function (and the corresponding optimal policy)! How do we do this? The answer is to instead define our **target as computed over a greedy policy which means using the Bellman Optimality Equation for our TD update!**

Then, we can run the Expected SARSA algorithm to learn the Q-function, extract the greedy policy from the Q-function, and repeat! The entire algorithm is as follows:

Algorithm 3 Q-learning Algorithm with SARSA

Initialize $\hat{Q}^*(s, a)$ arbitrarily
for each episode **do**
    Initialize s
    for each step of episode **do**
        Sample action: Take some action a from s, observe r, s'
        Update: $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$ (Bellman Optimality Equation now!!!)
        Go to next state: $s \leftarrow s'$
    end for
end for

Does this converge? (Pause.) Not necessarily if we don't have good enough exploration! For example, we may want to use epsilon greedy to sample actions. As long as we have good enough exploration, this algorithm will theoretically converge!!!

What about this algorithm, where we just choose actions randomly?

Algorithm 4 Q-learning Algorithm with SARSA, random version

Initialize $\hat{Q}^*(s, a)$ arbitrarily
for each episode **do**
    Initialize s
    for each step of episode **do**
        Randomly sample action from $Q(s, a)$: Take totally random actions a, observe r, s'
        Update: $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$ (Bellman Optimality Equation now!!!)
        Go to next state: $s \leftarrow s'$
    end for
end for

This algorithm will still converge! This is because Q-learning is an off-policy algorithm – we can learn the Q-function for the optimal policy even if the policy that generated our data is terrible or not the optimal policy!

5 On vs. off-policy???

Some of you may be wondering "why does the SARSA algorithm look so different in the homework vs. from lecture"? And how is the SARSA algorithm in the homework on-policy and the Q-learning algorithm in the homework off-policy?

In lecture, the SARSA algorithm we covered strictly focuses on a single pass of the inner loop of the homework pseudocode. Concretely:

Algorithm 5 SARSA Algorithm

Given: a set of transitions $\mathcal{T} = \{(s_0, a_0, r_0, s_0', a_0'), \dots\}$ collected from policy π
Initialize $Q^\pi(s, a)$ arbitrarily
for each $(s_i, a_i, r_i, s_i', a_i') \in \mathcal{T}$ **do**
    $\hat{Q}^\pi(s_i, a_i) \leftarrow (1 - \alpha)\hat{Q}^\pi(s_i, a_i) + \alpha(r + \gamma\hat{Q}^\pi(s_i', a_i'))$
end for
Update: $Q^\pi(s, a) \leftarrow \hat{Q}^\pi(s, a)$

Meanwhile, the inner loop of the homework pseudocode looks like this:

Algorithm 6 SARSA

Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}$
for each episode **do**
    Initialize s_t
    Choose a_t using ϵ-greedy policy w.r.t. $Q(s, a)$
    while s_t is not terminal **do**
        Take action a_t, observe r_t, s_{t+1}
        Choose a_{t+1} using ϵ-greedy policy w.r.t. $Q(s, a)$
        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big)$
        $s_t \leftarrow s_{t+1},\ a_t \leftarrow a_{t+1}$
    end while
end for

So, how is this on-policy and how is this related to the lecture algorithm? Note that we are always sampling our action a_t AND evaluating the target in the TD-learning update a_{t+1} from the same policy – the epsilon greedy policy with respect to $Q(s, a)$! This is similar to the SARSA presented in lecture for a single transition (again, we have this bag of transitions from policy π, and compute the target using the same policy π): **the target and transition's policy match because we are using a TD Update with Bellman Update, not a Bellman Optimality Update.** Thus, this algorithm is on-policy!

However, let's look at Q-learning: Note that the target and the transition's policies do not necessarily match! We choose our action a_t using an epsilon-greedy method while our TD Update using Bellman Optimality is done strictly with respect to the greedy action!!! You might be asking "well.... why choose a_t using an epsilon greedy policy at all? Can't we just use, say, a uniformly random policy, or just use a bunch of random trajectories we've collected before using potentially awful policies?" And the answer is yes! We can use a uniformly random policy – in fact, this algorithm will still theoretically converge to the optimal Q-value even if we always select a_t randomly (and random selection, in some very terrible sample complexity limit, will always get complete coverage of the desired state/action space). Thus, this algorithm is off-policy – the update and the transitions don't need to be over the same policy!

This is super different from SARSA – by definition, we need to choose a "good" a_t or else the a_{t+1} we do our TD learning update on will also be terrible (see the last line where $a_t \leftarrow a_{t+1}$). So, we need to constantly be resampling trajectories using our improved policy in order to correctly do policy improvement.

These differences will become more clear when you actually implement these algorithms! But just wanted to very concretely relate (1) what was in lecture, (2) what's in the homeowork, and (3) on vs. off-policy.

6 Summary of these TD-learning methods and shortcomings

In this class, we will consider Q-learning, SARSA, and Expected SARSA as three different algorithms:

- Q-learning directly learns Q^*. It is off-policy, model-free, and learns the optimal policy. **Cons:** Can be very unstable.
- SARSA learns Q^π given trajectories collected from π. It is on-policy and model-free. **Cons:** Can be very sample inefficient and may not converge to the optimal policy (when adding policy improvement).
- Expected SARSA learns $Q^{\pi'}$ for arbitrary π' given trajectories collected from π. It is off-policy and model-free. This is primarily a building block for Q-learning.

7 How do we scale up Q-learning?

In practice, we need extra tricks to scale up Q-learning to high-dimensional problems. Here are some tricks that we can use:

7.1 Experience replay

(Skip this in lieu of midterm review). Experience replay is a technique where we store a **replay buffer** of past trajectories and sample from this buffer to update the Q-function, instead of just updating on our collected trajectories from our current policy. Why is this useful? For Q-learning, during training, we may just have a terrible policy and thus terrible, correlated samples on our hands – this is bad! We somehow want to break these correlations and sample more fully from the entire set of our collected trajectories. **Thankfully, because Q-learning is an off-policy algorithm, we are totally free to sample from our replay buffer to find an optimal policy, even if the actions in our replay buffer are far from optimal!**

In practice, this modifies the Q-learning algorithm as follows:

Algorithm 7 Q-learning Algorithm with Experience Replay

Initialize $\hat{Q}^*(s, a)$ arbitrarily
Initialize replay buffer $\mathcal{D}$
for each episode **do**
    Initialize s
    for each step of episode **do**
        Greedily sample action from $Q(s, a)$**:** Take action $a = \arg\max_a Q(s, a)$, observe r, s'
        Store in replay buffer: $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s, a, r, s')\}$
        Sample from replay buffer: $(s, a, r, s') \sim \mathcal{D}$
        Update: $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$
        Go to next state: $s \leftarrow s'$
    end for
end for

7.2 Deep Q-learning

Spoiler alert: we use neural networks to approximate the Q-function! We'll explore this more next time (after the midterm)...

8 15 min midterm review

(A lot of cold calling.) Here is a summary of all the methods we've learned so far:

1. Bandits are a simple problem where we have a single state and multiple actions. We learned about the **ϵ-greedy policy and the UCB policy** – namely, there is a tradeoff between exploitation and exploration.
2. We learned about **CEM** as a way to minimize cross entropy without using gradients.
3. We can use CEM to solve **MPC** problems, where we have a model of the environment and we want to optimize a sequence of actions. The key idea is that we (1) use a model to get **sample** trajectories of our current policy (which just is a distribution over sequences of actions), (2) evaluate the rewards of the trajectories, (3) find the set of best trajectories, then (4) fit our action sequence to the action sequences that generated the best trajectories.
4. We learned the basics of **policy gradients** and how to use them to optimize a policy. We learned about the **REINFORCE** algorithm: aka we (1) sample trajectories, (2) compute the gradient of the expected reward with respect to the policy parameters (remember the policy gradient derivation?), and (3) update the policy parameters in the direction of the gradient.
5. We learned about **value functions and Q-functions**. We learned about the **Bellman Equation and Bellman Optimality Equation** and how to convert it into **(1) policy evaluation and iteration and (2) Value Iteration algorithm**. Notably, for both of these methods, we need access to the reward and transition functions.
6. We learned about **policy evaluation** and **policy improvement**. Broadly, policy evaluation is any algorithm that takes as input a policy and outputs a value function (or Q function). So Value Iteration is a policy evaluation method! Policy improvement is any algorithm that takes as input a value function and improves the policy using

this value function – the simplest version of policy improvement is the greedy policy improvement algorithm: $\pi(a \mid s) = \arg\max_a Q(s, a)$.

7. Alternating between policy evaluation and improvement is **policy iteration**.
8. We learned about **TD-learning methods**, which are model-free methods that learn the Q-function. We learned about the **SARSA** and **Expected SARSA** algorithms.
9. We learned about the **Q-learning algorithm**, which is an off-policy method that learns the Q-function – it's very similar to expected SARSA, but we add on a way to epsilon-greedily (or similar) extract a policy π from the Q-function.

Please go to the midterm review session to review these concepts in more detail!