CIS5200: Machine Learning Spring 2023

Lecture 24: Reinforcement Learning

Date: April 13, 2023 *Author: Surbhi Goel, Eric Wong*

Acknowledgements. These notes are heavily inspired by notes by Tengyu Ma (Stanford) and Sham Kakade (Harvard).

Disclaimer. These notes have not been subjected to the usual scrutiny reserved for formal publications. If you notice any typos or errors, please reach out to the author.

1 Reinforcement Learning

Today we will look at an online learning setup called *reinforcement* learning. Here instead of input examples and labels, we will instead have states and actions which lead to a reward. Our learner will be called an agent. For example, if we consider the task of programming a robot to walk, then the state would be the joint angles, the actions would be motor torques, the reward would be the average speed of walking.

More formally, each round t will look like the following:

1. Agent observes a state $s_t \in \mathcal{S}$
2. Agent takes action $a_t \in \mathcal{A}$
3. Agent receives reward $r_t \in \mathcal{R}$

The goal of the learner here is to find a *policy* $\pi : \mathcal{S} \to \mathcal{A}$ (or a probability distribution over actions $\Delta(\mathcal{A})$[1]) such that it maximizes reward.

This setup captures online learning setting, if we assume s_t is the instance x_t, the action is the prediction $\hat{y}_t$ and the reward is $-l(\hat{y}_t, y_t)$. Then learning a policy would be equivalent to learning a function that maps inputs to predictions. However, reinforcement learning is more general. As an example, in online learning, knowing y_t gives us access to knowing the loss of any function in the function class, whereas in this setup, the reward could reveal only partial information.

2 Bandits

Let us try and understand what partial information means through bandits. In the basic bandit, we assume that there is no state space, and the agent is just trying to choose one among many actions with the goal of maximizing reward. The name bandit comes from and English slang for slot machine.

[1] $\Delta(S)$ for any set S is used to denote the set of all probability distributions over set S

Let us consider an action space $\mathcal{A} = [k]$ of k actions where the reward function $R : \mathcal{A} \to \Delta(\{0, 1\})$ maps each action i to a probability distribution over $\{0, 1\}$. This is known as the k-armed bandit problem. Each time we draw an action i, we get a reward $r_i \sim R(i)$. We do not get to see what the reward would have been for the other arms had we pulled them.

Maximizing reward reduces to the following dilemma, should we *explore* more to find a better action or *exploit* the best action we have so far. More formally, suppose we have pulled each arm a few times and have gotten estimates $\hat{p}_i$ of $\mathbb{E}[R(i)]$. Now we could either exploit by selecting the arm with the highest $\hat{p}$, or explore by trying the other arms to improve our estimates $\hat{p}$. *Think about why exploration is important.*

There is a generalization of bandits to contextual bandits where we add back states to the bandit problem, and now there is an independent bandit problem for each state.

3 Markov Decision Process (MDP)

We could make this more challenging, by assuming that the action in each state actually changes our state. This is somewhat like a contextual bandit problem, but more complicated, since in a contextual bandit, the action in a state did not change the state.

Formally, an MDP is $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$ where

- $\mathcal{S}$ is the state space
- $\mathcal{A}$ is the action space
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the transition model,

  $$T(s, a, s') = \mathcal{P}(s'|s, a),$$

  the probability of transitioning to state s' conditioned on taking action a in state s.
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function where $R(s, a)$ specifies the reward for taking action a in state s.
- $\gamma \in [0, 1]$ is the discount factor

The dynamics of an MDP start with some initial state s_0 and the agent chooses a_0. Then $s_1 \sim T(s, a, \cdot)$ and then the process repeats.

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \cdots$$

The total reward is given by

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \gamma^3 R(s_3, a_3) + \ldots = \sum_{i=0}^{\infty} \gamma^i R(s_i, a_i).$$

Here γ discount factor is applied such that reward at time t has a factor of γ^t. This makes near-term reward more important than way in the future reward. We will assume a deterministic policy $\pi : \mathcal{A} \to \mathcal{A}$.

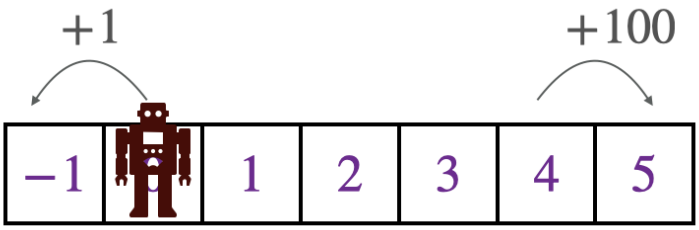

Figure 1: A robot walking on a line.

Example: Robot on a line. Consider making a robot walk on a line as in Figure 1. The robot starts at position 0. If you tell the robot to move left or right, it obeys you 99% of the time, this implies the transition functions T satisfies

$$T(s, \text{left}, s') = \begin{cases} 0.99 & \text{if } s' = s = -1 \\ 0.99 & \text{if } s' = s - 1 \text{ and } s \neq -1 \\ 0.01 & \text{if } s' = s = 5 \\ 0.01 & \text{if } s' = s + 1 \text{ and } s \neq 5 \\ 0 & \text{otherwise.} \end{cases}$$

$$T(s, \text{right}, s') = \begin{cases} 0.01 & \text{if } s' = s = -1 \\ 0.01 & \text{if } s' = s - 1 \text{ and } s \neq -1 \\ 0.99 & \text{if } s' = s = 5 \\ 0.99 & \text{if } s' = s + 1 \text{ and } s \neq 5 \\ 0 & \text{otherwise.} \end{cases}$$

Here reward satisfies,

$$R(s, a) = \begin{cases} 1 & \text{if } s = 0, a = \text{left} \\ 100 & \text{if } s = 4, a = \text{right} \\ 0 & \text{otherwise.} \end{cases}$$

Value function. We define the value function of a policy $\pi : \mathcal{S} \to \mathcal{A}$ at state s

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i R(s_i, a_i) \middle| s_0 = s, \pi\right].$$

This is the expected sum of discounted rewards upon starting at s and taking actions according to policy π. This can be rewritten as,

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V^{\pi}(s').$$

Here the first term is the reward at the current state and the second is expected sum of the rewards over the choice of transitioned states. These are known as the **Bellman equations**. If we could write these equations for all $s \in \mathcal{S}$, since they are linear, we can solve them to find $V^{\pi}(s)$.

The optimal value function is defined as,

$$V^*(s) = \max_\pi V^\pi(s) = \max_a \left[R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right].$$

Q function. Similar to value function, we can define the action-value Q function of a policy $\pi : \mathcal{S} \to \mathcal{A}$ at state s taking action a as

$$Q^\pi(s,a) = \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i R(s_i, a_i) \middle| s_0 = s, a_0 = a.\pi \right].$$

The Bellman equations corresponding to this are,

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') Q^\pi(s', \pi(s'))$$

The optimal Q function is defined as,

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') \max_{a'} Q^*(s',a').$$

Unlike in V^*, these equations are not linear but there is a theorem that says they have a unique solution!

Observe that,

$$V^\pi(s) = Q(s, \pi(s)) \text{ and } V^*(s) = \max_a Q^*(s,a).$$

Optimal Policy. Using $Q^*(s,a)$ we can get the optimal policy as,

$$\pi^*(s) = \arg\max_a Q^*(s,a).$$

the action that gets us the maximum expected reward.

Iteration procedures. If we know T and R, then we can solve for optimal Q^*, V^* using an iterative procedure for solving the equations. Note that there is no learning here. Here is an example of value iteration for Q-function:

Algorithm 1: Value Iteration for Q function

```
Initialize Q^1(s,a) = 0 for all s ∈ S, a ∈ A
for t = 1, 2, ... do
    for s ∈ S, a ∈ A do
        Update Q^{t+1}(s,a) = R(s,a) + γ Σ_{s'∈S} T(s,a,s') max_{a'} Q^t(s',a')
    end
end
```

Let us run this algorithm on our robot example for $\gamma = 0.9$. Initially we have,

$$Q^1_{\text{left}} = \begin{bmatrix} 0,0,0,0,0,0,0 \end{bmatrix} \qquad Q^1_{\text{right}} = \begin{bmatrix} 0,0,0,0,0,0,0 \end{bmatrix}$$

For $t = 1$,

$$Q^2_{\text{left}} = [0, 1, 0, 0, 0, 0, 0] \qquad Q^2_{\text{right}} = [0, 0, 0, 0, 0, 100, 0]$$

For $t = 2$,

$$Q^2_{\text{left}} = [0.009, 1, 0.891, 0, 0.9, 0, 89.1] \qquad Q^2_{\text{right}} = [0.891, 0, 0.009, 0, 89.1, 100, 0.9]$$

After many steps,

$$Q^T_{\text{left}} = [276.38, 277.73, 307.79, 342.65, 381.46, 423.66, 471.16]$$

$$Q^T_{\text{right}} = [306.75, 341.18, 379.82, 422.84, 470.72, 524.04, 424.52]$$

This gives the optimal policy as

$$\pi^* = [\text{right}, \text{right}, \text{right}, \text{right}, \text{right}, \text{right}, \text{left}]$$

Here's a collab to play around with this example https://colab.research.google.com/drive/1LgFt5WgAHVKMxms0tQV3zNOxjrc6M18f?usp=sharing.

Estimating T and R. In general, we may not have access to T and R but rather a sequence of (s_t, a_t, r_t, s'_t). We can use these to estimate

$$\hat{T}(s, a, s') = \frac{\#(s, a, s')}{\#(s, a)} \qquad \hat{R} = \frac{\sum r | s, a}{\#(s, a)}.$$

To avoid dividing by 0, we usually add a Laplace correction,

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

Once we have these, we can run value iteration as above.

4 Q-learning

In general, it may not always be possible to estimate T and R. For example, our sequence of observations may not cover all possible state, action, state triplets, or these may even be continuous spaces. If we can still simulate sequences, we can still make progress with an algorithm called Q-learning.

Q-learning is a way to estimate the Q function when the agent is allowed to simulate taking single steps in the environment. It is an iterative process with a step size with similarities to gradient descent optimization procedures.

Algorithm 2: Q-Learning

Initialize $Q(s, a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize s randomly from $\mathcal{S}$
for $t = 1, 2, \ldots$ **do**
  Update $a = \texttt{select\_action}(s, Q)$
  Update $r, s' = \texttt{execute}(a)$
  Update $Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(R(s, a) + \gamma \max_{a'} Q(s', a'))$
  Update $s = s'$
end

Note that this update can be rewritten to look like a gradient descent update:

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

The update on the right can be viewed as the difference between the current estimated value of taking action a in state s, $Q(s,a)$, and the one-step simulated value of taking action a in s.

This also has a flavor of dynamic programming: we are using the previously computed solutions of $Q(s',a')$ to update our current solution for $Q(s,a)$.

`select_action` Q-learning requires a way to select actions based on the current estimate of Q and the current state s. If Q was perfectly estimated, i.e. $Q = Q^*$, then the optimal action is simply the optimal policy $\pi(s) = \arg\max_a Q^*(s,a)$. However, while we are running Q-learning, we can't expect Q to be close to optimal, so we need a way to allow for some exploration. A typical strategy is the ϵ-greedy strategy:

$$\texttt{select\_action}(s) = \begin{cases} a = \arg\max_a Q(s,a) & \text{with probability } 1-\epsilon \\ a \sim \text{Uniform}(\mathcal{A}) & \text{with probability } \epsilon \end{cases}$$

Under fairly weak conditions, Q-learning *guaranteed* to converge to the optimal Q function. In particular, any exploration strategy is fine as long as it tries every action infinitely often on an infinite run. This avoids premature convergence to a bad action.

However, even though it is guaranteed to converge, Q-learning can require a large number of iterations or samples to converge. Going back to the robot example, if a robot has a choice between going to the left or to the right from the initial state, the robot can quickly find a reward for left but will not immediately get any reward for going right. It needs to move down the line at least 5 steps before learning that going to the right was a good choice. If we take $\alpha - 1$ and $\gamma = 0.9$, then

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

As the robot goes to the right, the Q values remain zero even if it goes right 4 steps. On the 5th step, the Q value of the 4th state finally gets updated to reflect the reward at the end of the line.

$$Q_{\text{right}} = [0, 0, 0, 0, 0, 0, 100, 0]$$

However, as the robot goes left, these Q values don't get updated. These values only get updated when the robot goes right again, this time from the 3rd state to the 4th state.

$$Q_{\text{right}} = [0, 0, 0, 0, 90, 100, 0]$$

and so on. Thus, to propagate updates for the Q function back to the origin, we need to go right down the line 5 times before we have learned that there is value in going right from the origin.

Function approximation with neural network Up to this point we've assumed that the state and action spaces are discrete spaces that can be represented in tabular form. If the state and action spaces are large or continuous, we'd like a more efficient way to represent the Q values. One popular way is to use a neural network to represent Q.

To learn Q, we can treat this as a regression problem using the squared Bellman error. The Bellman error is the same as the "gradient step" update from Q-learning, so if this error is small, then this is equivalent to Q-learning having converged.

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

Neural networks don't always have nice properties in Q learning. It can be fairly unstable, forgetting past rewards, and not exploring enough. There's a bunch of tricks people have developed to combat these challenges (replay buffers, sliding windows, etc.).