COMS4047A - Reinforcement Learning
Lab 2 - Model-Free Learning

Shahil Mawjee

Overview

The lab will focus on the model-free learning methods.

For this lab we will use two environments which are contained within the OpenAI Gym python package:

- Blackjack (repo link) - `gym.make('Blackjack-v0')`
- Cliffwalking (repo link) - `gym.make('CliffWalking-v0')`

Goals:

- Understand the difference between dynamic programming and model-free methods.
- Understand Monte-Carlo methods.
- Understand Temporal Difference methods.

Why can't we always rely on Policy Iteration or Value Iteration ?

At each state, we look ahead one step at each possible action and next state.

1. We can only do this because we have a perfect model of the environment. (the transition matrix P). In most applications, this assumption does not hold. We would like an algorithm that can learn from *experience*, i.e. by simply interacting with the environment.
2. DP methods are polynomial in the number of states. Many real world applications have a very large state space, making DP methods unusable.

1 Monte-Carlo methods

Monte Carlo (MC) methods can learn directly from experience collected by interacting with the environment. An episode of experience is a series of (State,

Action, Reward, Next State) tuples.

MC methods work based on episodes. We sample episodes and make updates to our estimates at the end of each episode. MC methods have high variance (due to lots of random decisions within an episode) but are unbiased.

Environment - Blackjack

For the MC methods we will use the OpenAI Gym Blackjack-v0 environment. Blackjack is a card game where the goal is to obtain cards that sum to as near as possible to 21 without going over. They're playing against a fixed dealer.

Face cards (Jack, Queen, King) have point value 10. Aces can either count as 11 or 1, and it's called *'usable'* at 11. This game is placed with an infinite deck (or with replacement). The game starts with each (player and dealer) having one face up and one face down card.

The player can request additional cards (hit=1) until they decide to stop (stick=0) or exceed 21 (bust).

After the player sticks, the dealer reveals their facedown card, and draws until their sum is 17 or greater. If the dealer goes bust the player wins. If neither player nor dealer busts, the outcome (win, lose, draw) is decided by whose sum is closer to 21. The reward for winning is +1, drawing is 0, and losing is -1.

The observation of a 3-tuple of: the players current sum, the dealer's one showing card (1-10 where 1 is ace), and whether or not the player holds a usable ace (0 or 1).

1.1 Prediction

Given a policy π, can we find the corresponding value function v_π ?

Sample episodes of experience and estimate $V(s)$ to be the return received from that state onwards averaged across all of your experience. The same technique works for the action-value function $Q(s, a)$. Given enough samples, this is proven to converge.

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
            Append G to $Returns(S_t)$
            $V(S_t) \leftarrow$ average$(Returns(S_t))$

Figure 1: Source - Sutton & Barto Section 5.1, page 92

Exercise

1. Create a policy that sticks if the player score is >= 20 and hits otherwise. Give an observation the policy should return the correct action. Use `blackjack_sample_policy` function.

2. Complete the `mc_prediction` function. This function takes in a policy function, the gym environment and the number of episodes. Returns a dictionary that maps from state to value.

3. Run `mc_prediction` function with `blackjack_sample_policy` and the Blackjack environment. With 10 000 episodes and 500 000 episodes. Plot your results using `blackjack_plot_value_function` function.

4. Is there any difference between the two runs? if yes, explain why?

1.2 Control

The idea is the same as for Dynamic Programming. Use MC Policy Evaluation to evaluate the current policy then improve the policy greedily. The problem: How do we ensure that we explore all states if we do not know the full environment?

Solution to exploration problem: Use epsilon-greedy policies instead of full greedy policies. When making a decision act randomly with probability epsilon. This will learn the optimal epsilon-greedy policy.

Since we are doing control, we will want to estimate the value of (state, action) pairs. We will therefore be working with $Q(s, a)$ and not $V(s)$

Side note: It's important to break ties arbitrarily when doing control. This

is especially important when you initialize all Q or V array to all zeros. If you don't break ties arbitrarily, you will end up always choosing the same action!

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

Figure 2: Source - Sutton & Barto Section 5.4, page 101

Exercise

1. Implement the `argmax` function that break ties randomly.
2. Implement the `make_epsilon_greedy_policy` function, which return a policy function that takes in the observation.
3. Complete `mc_control_epsilon_greedy` function.
4. Convert the $Q(s, a)$ to $V(s)$ by taking the max action a value for each state s. Then plot the values using `blackjack_plot_value_function`

2 Temporal Difference (TD) learning

TD-Learning is a combination of Monte Carlo and Dynamic Programming ideas. Like Monte Carlo, TD works based on samples and doesn't require a model of the environment. Like Dynamic Programming, TD uses bootstrapping to make updates. At the core of TD learning is the following update rule

$$V(s_t) \leftarrow V(s_t) + \alpha \Big[R_{t+1} + \gamma V(S_{t+1}) - V(s_t) \Big]$$

(to contrast with MC updates)

$$V(s_t) \leftarrow V(s_t) + \alpha\Big[G_t - V(s_t)\Big]$$

In other words, for every update, the target is actually based on our predicted value for the next state, $R_{t+1} + \gamma V(S_{t+1})$, instead of being the sum of observed rewards, G_t

Why is this valid? Simply because

$$\begin{aligned} v_\pi(s) &= \mathbb{E}[G_t|St = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \end{aligned}$$

Environment - Cliff Walking

To find out more information about the the Cliff walking environment refer to the github repo or the text book on page 132.

2.1 Prediction

Here is the pseudo-code for doing policy evaluation with TD

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 3: Source - Sutton & Barto Section 6.1, page 120

2.2 Control

Let's skip prediction for TD methods and go straight to control.

Q-learning

Arguably the most famous TD algorithm is Q-Learning. Q-learning is an off-line method, since the target update does not depend on the behaviour policy (because of the max operator).

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 4: Source - Sutton & Barto Section 6.5, page 131

SARSA

The on-line version of Q-Learning is known as SARSA (which stands for State, Action, Reward, State, Action). Notice that in the following pseudo-code, the action selected in the target update is the same as the action used in the next time step.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figure 5: Source - Sutton & Barto Section 6.4, page 130

Exercise

1. Complete `q_learning` function. The function takes in the gym environment and the number of episodes. It will return a dictionary mapping state to action-values and `stats` is an `EpisodeStats` object with two numpy arrays for `episode_lengths` and episode_rewards. The `stats` just keep track of the rewards acquired for each step.

2. Complete `SARSA` function.The function takes in the gym environment and the number of episodes. It will return a dictionary mapping state to action-values and `stats` is an `EpisodeStats` object with two numpy arrays for `episode_lengths` and episode_rewards. The `stats` just keep track of the rewards acquired for each step.

3. Plot `td_plot_episode_stats` and `td_plot_values` for both TD methods.

4. Refer to the value plots, notice the different policy each method achieved. Why are the policies different?

5. How will Q-learning and SARSA compare if you evaluate the learned policies with ϵ=0?

6. What will happen if we train Q-learning and SARSA with ϵ=0?

Submission

The lab will be automatically graded. The marker will call the functions directly. Ensure your python file is named `lab_2.py` and that the method names and parameters remain the same.

Once your have completed the lab, zip your code file and the plots. Submit your zip file to Moodle.

Your file structure when submitting should look like:
submission.zip
–lab_2.py

and NOT:
submission.zip
–submission
—-lab_2.py