Monte Carlo Methods

Story So Far...

- In the previous chapter, we learned how to compute the optimal policy using two dynamic programming methods
  - value iteration
  - policy iteration

- Dynamic programming can be used when the model dynamics is known.
  - transition probability
  - reward

- What is we do not know the model dynamics?
  - In that case, we cannot use dynamic programming.

- We need a model-free learning method when we do not know the model dynamics.
- The Monte Carlo method is one of the model-free methods.

The Monte Carlo Method

- A statistical technique used to find an approximate solution through sampling.

- Suppose we have a random variable X, for which we do not know the probability distribution. We want to find out its expectation $E(X)$.

- If we know the probability distribution, then $E(X)$ can be calculated as:

$$E(X) = \sum_{i=1}^{N} x_i p(x_i)$$

- If we do not know the probability distribution, then we can just find out by sampling values of X for some N times. We will approximate the expectation with the mean of sampled values. The approximation gets better when N is larger.

$$\mathbb{E}_{x\sim p(x)}[X] \approx \frac{1}{N}\sum_{i} x_i$$

Prediction Task and Control Task

- Prediction Task
  - With policy π given, we try to predict the value function or Q function using the policy.
  - Why? because we want to evaluate the policy.
  - What is a good policy? One that gets a good return for the agent.
  - How can we get the return? From the Q function
  - Thus, by predicting a Q function, we predict the (expected) return, and that will evaluate the policy π.
  - In prediction task, we don't make any change to the given policy.

- Control Task
  - We are not given any policy here. We start off with a random policy.
  - Iteratively, we make changes to the policy, hoping to find the optimal policy.

Monte Carlo Methods

Prediction Task using the Monte Carlo method

The Value Function

- Here we would like to calculate a value function for an environment where model dynamics is not known.

- Definition of a value function is:

$$V^{\pi}(s) = \underset{\tau \sim \pi}{\mathbb{E}} [R(\tau) | s_0 = s]$$

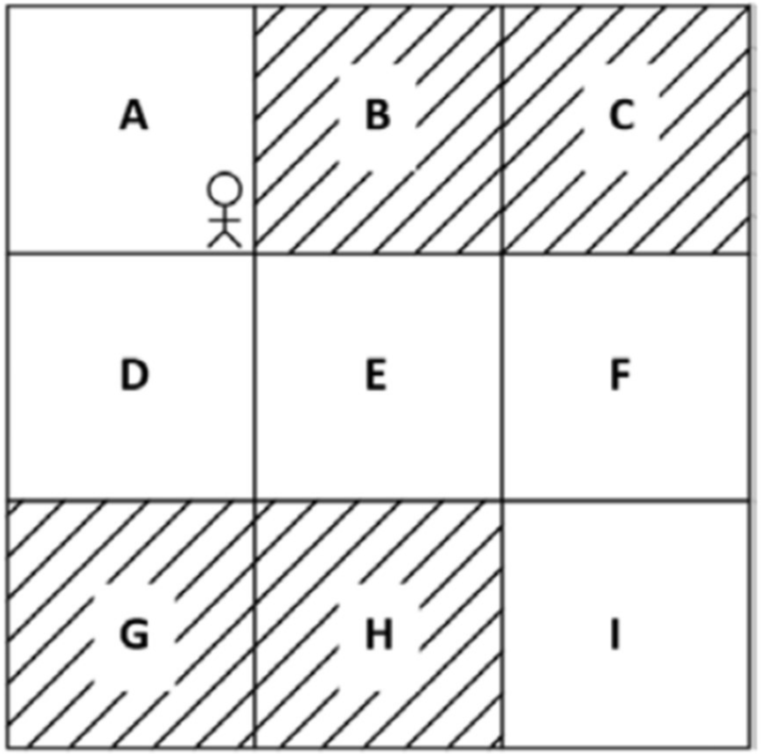
  – It means the expected return when the agent starts from state s and follows policy π.

- We can approximate the value of the state using the Monte Carlo method.
- We sample episodes following the given policy π for some N times and compute the value of the state as the average return across sampled episodes.

$$V(s) \approx \frac{1}{N} \sum_{i=1}^{N} R_i(s)$$

The Grid World Again

- We consider the 3x3 Grid World again.
- The agent has a choice of "moving right" and "moving down".
- Here, the agent obtains +1 reward when it visits an unshaded state, and obtains -1 reward when it visits a shaded state.
- The episode ends when the agent arrives at state I.



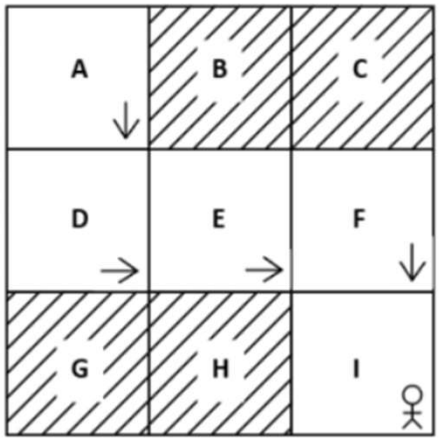
Monte Carlo in Grid World

- Suppose we have an environment where the transition probability is like the following.
  - If agent moves "down" in state A, it moves to state D 80% of the time, and it moves to state B 20% of the time.
  - For all other states, the agent successfully moves according to its intention.
    - If the agent moves right, it will move to the right state 100% of the time.

- In this environment, we are going to have a policy π that does the following.
  - In State A, B, C, and F, the agent selects "down".
  - In State D, E, G, and H, the agent selects "right".

Monte Carlo in Grid World

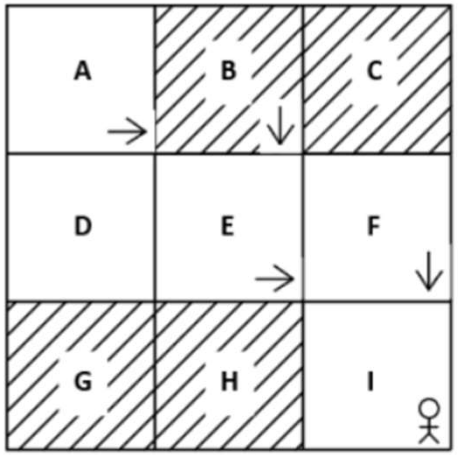
- We generate an episode τ_1 using the policy π.
  - In this episode, the agent moves to state D by "moving down" in state A.
  - The return $R_1(A)$ is +1+1+1+1 = 4.

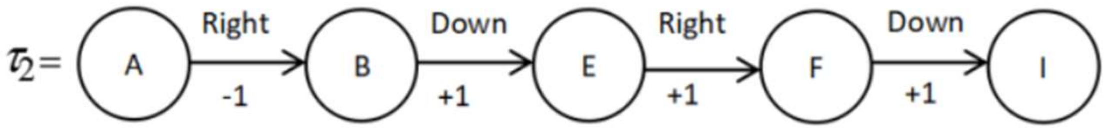


$$\tau_1 = A \xrightarrow[+1]{\text{Down}} D \xrightarrow[+1]{\text{Right}} E \xrightarrow[+1]{\text{Right}} F \xrightarrow[+1]{\text{Down}} I$$

Monte Carlo in Grid World

- We generate another episode τ_2 using the policy
  - In this episode, the agent moves to state B by "moving down" in state A.
  - The return $R_2(A)$ is -1+1+1+1 = 2.





- We generate another episode τ_3 using the policy
  - In this episode, the agent moves to state D by "moving down" in state A.
  - The return $R_3(A)$ is +1+1+1+1 = 4.

Monte Carlo in Grid World

- Now, we estimate value of state A using the three episodes.
- We let $V(A)$ to be the average return of the three sampled episodes.

$$V(A) \approx \frac{1}{N}\sum_{i=1}^{N} R_i(A)$$

- In this example, $V(A) = \frac{4+2+4}{3} = 3.3$

- What is the "real" value of state A according to policy π?
  - If the agent moves to state B from state A, $V(A) = 2$.
  - If the agent moves to state D from state A, $V(A) = 4$.
  - Since the agent moves to D in 80% of the time, $V(A) = 0.8 \times 4 + 0.2 \times 2 = 3.6$.

- Similarly, we can compute the value of other states.
- The accuracy of estimation is improved when number of samples is larger.

Monte Carlo Prediction Algorithm

1. Let total_return(s) be the sum of return of a state across several episodes and $N(s)$ be the number of times a state is visited across several episodes. Initialize total_return(s) and $N(s)$ as zero for all the states. The policy π is given as input.

2. For M number of iterations:
    1. Generate an episode using the policy π.
    2. Store all rewards obtained in the episode in the list called **rewards**.
    3. For each step t in the episode:
        1. Compute the return of state s_t as $R(s_t)$ = sum(rewards[:t])
        2. Update total return of state s_t as total_return(s_t) = total_return(s_t) + $R(s_t)$
        3. Update the counter as $N(s_t) = N(s_t) + 1$

3. Compute the value of state by taking the average, that is:

$$V(s) = \frac{\text{total\_return(s)}}{N(s)}$$

Types of Monte Carlo Prediction

- First-visit Monte Carlo
  - If a state is visited multiple times in an episode, only consider the first visit to that state when calculating the value function.

- Every-visit Monte Carlo
  - If a state is visited multiple times in an episode, consider all visits to the state when calculating the value function.

First-Visit Monte Carlo

1. Let total_return(s) be the sum of return of a state across several episodes and $N(s)$ be the number of times a state is visited across several episodes. Initialize total_return(s) and $N(s)$ as zero for all the states. The policy π is given as input.

2. For M number of iterations:
    1. Generate an episode using the policy π.
    2. Store all rewards obtained in the episode in the list called **rewards**.
    3. For each step t in the episode:

        If the state s_t is occurring for the first time in the episode:
        1. Compute the return of state s_t as $R(s_t)$ = sum(rewards[:t])
        2. Update total return of state s_t as total_return(s_t) = total_return(s_t) + $R(s_t)$
        3. Update the counter as $N(s_t) = N(s_t) + 1$

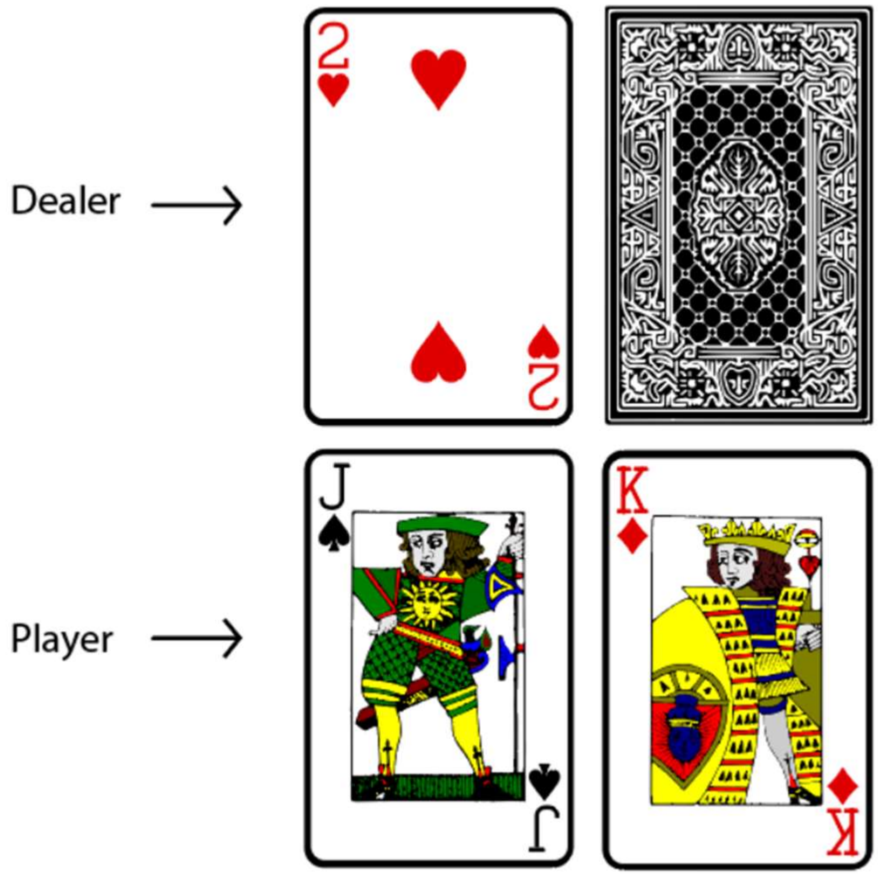
3. Compute the value of state by taking the average

Implementing the Monte Carlo Prediction Method

- Our environment: Blackjack
- Understanding the game of Blackjack
  - Blackjack is a card game that is played between a player and a dealer.
  - The goal of the player is to have the value of the sum of all the cards be 21, or a larger value than the sum of the dealer's cards while not exceeding 21.
  - The value of the cards Jack (J), Queen (Q), and King (K) is considered as 10.
  - The value of the Ace (A) can be 1 or 11, depending on the player's choice.
  - The value of the rest of the cards (2 to 10) is their face value.

Blackjack - Rules

- Initially, a player is given two cards. Both cards are face up.
- Similarly, the dealer is also given two cards. One of the card is face up, and the other is face down.



Blackjack - Rules

- The player plays first.
- The player has two actions - Hit or Stand.
- If the player selects "Hit", an additional card is given to the player.
  - The player can select "Hit" multiple times until deciding to select "Stand".
  - If the sum exceeds 21 after "Hit", then the player automatically loses and the game ends there. This is called "bust".
- If the player selects "Stand", the player stops getting more cards.

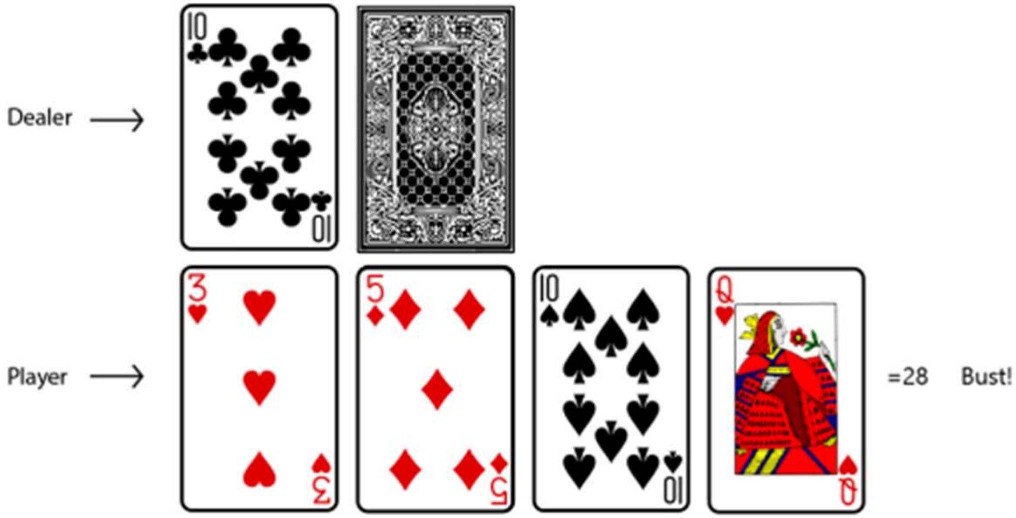
- Then, the dealer plays.
- The dealer has a fixed policy. If the sum is less than 17, the dealer selects "Hit". Otherwise, the dealer selects "Stand".

- Once the player and the dealer is done playing, their cards are compared to select the winner.

Blackjack - Rules

- If the player "busts" (the sum exceeds 21), then the player loses.
- If the player's sum is less than or equal to 21 and the dealer "busts", the player wins.
- If the player's sum is larger than the dealer's sum, the player wins.
- If the player's sum is smaller than the dealer's sum, the player loses.
- If the player's sum is equal to the dealer's sum, the game is a draw.

- Reward for a Win: +1
- Reward for a Draw: 0
- Reward for a Loss: -1



the player goes bust

Blackjack - Rules

- "Usable" Ace
  - The Ace (A) card can be used as 1 or 11.
  - If the player holds an Ace card that is being used as 11 without going bust, we say that the player has a "usable" ace.
    - The player might select "hit" even with a large number.
  - e.g.) If the player has an ace and 5, the ace here is the "usable" ace.

Modeling Blackjack for Reinforcement Learning

- Although the game involves two players (the player and the dealer), this is a single-agent environment because the dealer has a deterministic policy.

- States
  - Player's sum
  - Dealer's face-up card
  - Whether the player has a usable ace or not

```
(15, 9, True)
```

- Actions
  - Stand (0), Hit (1)

- Rewards
  - Win (+1), Draw (0), Loss (-1)

Every-visit Monte Carlo Prediction for the Blackjack Game

- Create the environment
  - We are going to use the pandas library.

```
import gym
import pandas as pd
from collections import defaultdict
```

```
env = gym.make('Blackjack-v0')
```

- Define a policy
  - To run a Monte Carlo prediction, we need an input policy.
  - In our policy, we will choose Stand if our sum is larger than 19. Otherwise, we will choose Hit.

```
def policy(state):
    return 0 if state[0] > 19 else 1
```

Every-visit Monte Carlo Prediction for the Blackjack Game

- Define a function that generates a single episode
  - Perform actions based on the policy
  - Record (state, action, reward) for each state it visits

```python
def generate_episode(policy):
    episode = []
    state = env.reset()
    for t in range(num_timesteps):
        action = policy(state)
        next_state, reward, done, info = env.step(action)
        episode.append((state, action, reward))
        if done:
            break
        state = next_state

    return episode
```

Every-visit Monte Carlo Prediction for the Blackjack Game

- Run multiple episodes and record the total return and N (number of times the state is visited) for all states.
- Value of a state is the total_return of the state divided by N of the state.
- We can use the pandas dataframe to conveniently organize the value table.

```
total_return = defaultdict(float)
N = defaultdict(int)
num_iterations = 500000
for i in range(num_iterations):
    episode = generate_episode(policy)
    states, actions, rewards = zip(*episode)
    for t, state in enumerate(states):
        R = (sum(rewards[t:]))
        total_return[state] = total_return[state] + R
        N[state] = N[state] + 1

total_return = pd.DataFrame(total_return.items(), columns=['state', 'total_return'])
N = pd.DataFrame(N.items(), columns=['state', 'N'])
df = pd.merge(total_return, N, on='state')
df['value'] = df['total_return'] / df['N']
```

Every-visit Monte Carlo Prediction for the Blackjack Game

- As the result of Monte Carlo prediction, we get the value for each state.
  - The left only shows the first 10 lines in the table.
  - From the right, we can see that if the player's sum is 21, the approximated expected return is 0.938015, which is a very high value.
  - If the player's sum is 5 and the dealer's face-up card is an Ace, than the value of the state is -0.615741, a low value.

```
df.head(10)
```

✓ 0.2s

	state	total_return	N	value
0	(15, 6, False)	-3213.0	4973	-0.646089
1	(19, 6, False)	-4071.0	5504	-0.739644
2	(9, 10, False)	-3431.0	6210	-0.552496
3	(20, 10, True)	1696.0	3734	0.454205
4	(16, 5, False)	-3362.0	5117	-0.657026
5	(17, 5, False)	-3589.0	5282	-0.679477
6	(18, 5, False)	-3772.0	5447	-0.692491
7	(6, 10, False)	-1555.0	2803	-0.554763
8	(10, 10, False)	-1529.0	7586	-0.201555
9	(18, 10, False)	-15842.0	21890	-0.723709

```
df[df['state']==(21,9,False)]['value']
```

✓ 0.3s

```
15    0.938015
Name: value, dtype: float64
```

```
df[df['state']==(5,1,False)]['value']
```

✓ 0.8s

```
243    -0.615741
Name: value, dtype: float64
```

First-visit Monte Carlo Prediction for the Blackjack Game

- When implementing the First-visit Monte Carlo prediction, we can just change one line in the code implementing the Every-visit MC prediction.

```
total_return = defaultdict(float)
N = defaultdict(int)
num_iterations = 500000
for i in range(num_iterations):
    episode = generate_episode(policy)
    states, actions, rewards = zip(*episode)
    for t, state in enumerate(states):
        if state not in states[0:t]:
            R = (sum(rewards[t:]))
            total_return[state] = total_return[state] + R
            N[state] = N[state] + 1
```

Incremental Mean Updates

- In the previous example code, we used the "arithmetic mean" to calculate the average return of the state across multiple episodes.

$$V(s) = \frac{\text{total\_return}(s)}{N(s)}$$

- Another way is to use incremental mean

$$V(s_t) = V(s_t) + \alpha\big(R_t - V(s_t)\big)$$

  - where $\alpha = \frac{1}{N(s_t)}$

- The incremental mean can be used when we prefer using returns from the latest episodes compared to the earlier episodes.
  - e.g.) non-stationary environment

Monte Carlo Prediction of the Q Function

- Previously, we used Monte Carlo prediction to predict the value function.
- Similarly, we can use Monte Carlo prediction to predict the Q function.

- MC Prediction of the Q Function
  - We generate multiple episodes using the given policy π.
  - We calculate total_return(s,a), the sum of the return of the state-action pair across multiple episodes.
  - We also calculate $N(s, a)$, the number of times the state-action pair is visited across multiple episodes.
  - Then, we calculate the Q value for a state-action pair as:

$$Q(s,a) = \frac{\text{total\_return}(s,a)}{N(s,a)}$$

Monte Carlo Prediction of the Q Function

- The algorithm for predicting the Q function using the Monte Carlo method

1. Let total_return(s, a) be the sum of the return of the state-action pair across several episodes and $N(s, a)$ be the number of times a state-action pair is visited across several episodes. Initialize total_return(s, a) and $N(s, a)$ for all state-action pairs to zero. The policy π is given as input.
2. For M number of iterations:
    1. Generate an episode using policy π.
    2. Store all rewards obtained in the episode in the list called rewards.
    3. For each step t in the episode:
        1. Compute the return for the state-action pair, $R(s_t, a_t)$ = sum(rewards[t:])
        2. Update the total return of the state-action pair, total_return(s_t, a_t) += $R(s_t, a_t)$
        3. Update the counter as $N(s_t, a_t)$ += 1
3. Compute the Q function by taking the average

$$Q(s,a) = \frac{\text{total\_return}(s,a)}{N(s,a)}$$

Monte Carlo Prediction of the Q Function

- Every-visit vs. First-visit
  - Similar to when predicting the value function, we have two types of MC: first-visit MC and every-visit MC.

- Arithmetic mean vs. Incremental mean
  - Incremental mean can be used instead of arithmetic mean

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\big(R_t - Q(s_t, a_t)\big)$$

Monte Carlo Methods

Control Task using the Monte Carlo method

Monte Carlo Control

- In the previous section, we learned how to predict the value function and the Q function using the Monte Carlo method.

- In the control task, we use the predicted functions to find the optimal policy.

- The procedure for Monte Carlo control
  - Start with a random policy
  - We predict the Q function of the random policy
  - From the Q function, we extract a new policy by selecting an action in each state that has the maximum Q value. That is:

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

  - We predict the Q function using the new policy
  - From the Q function, we extract a new policy
  - Repeat the iteration until we think we have found the optimal policy

$$\pi_0 \rightarrow Q^{\pi_0} \rightarrow \pi_1 \rightarrow Q^{\pi_1} \rightarrow \pi_2 \rightarrow Q^{\pi_2} \rightarrow \pi_3 \rightarrow Q^{\pi_3} \rightarrow \cdots \rightarrow \pi^* \rightarrow Q^{\pi^*}$$

Monte Carlo Control Algorithm

1. Let total_return(s, a) be the sum of the return of a state-action pair across several episodes and $N(s, a)$ be the number of times a state-action pair is visited across several episodes. Initialize total_return(s,a) and $N(s, a)$ for all state-action pairs to zero and initialize a random policy π.
2. For M number of iterations:
   1. Generate an episode using policy π.
   2. Store all rewards obtained in the episode in the list called rewards.
   3. For each step t in the episode:

      If (s_t, a_t) is occurring for the first time in the episode:
      1. Compute the return of a state-action pair, $R(s, a)$=sum(rewards[t:])
      2. Update the total return of the state-action pair, total_return(s_t, a_t) += $R(s_t, a_t)$
      3. Update the counter as $N(s_t, a_t)$ += 1
      4. Compute the Q value by taking the average $Q(s_t, a_t) = \dfrac{\text{total\_return}(s_t, a_t)}{N(s_t, a_t)}$
   4. Compute the new updated policy π using the Q function:

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

Types of Monte Carlo Control

- On-policy control
  - The agent behaves using one policy π, and tries to improve the same policy π.

- Off-policy control
  - The agent behaves using one policy b and tries to improve a different policy π.

On-Policy Monte Carlo Control

- In on-policy MC control, we iteratively update policy π.
  - Initially, π is a random policy.
  - Generate multiple episodes using π, and calculate Q values for state-action pairs.
  - From the Q function, extract a new policy π.
  - Repeat.

- The problem with this procedure is that some state-action pairs may never be experienced.
  - If the agent never explores a particular action in a state, we never know whether it is a good action or not.

- In order to explore new state-action pairs, we must choose actions other than one that has the maximum Q value.
  - This is called exploration.

On-Policy Monte Carlo Control

- One way to include exploration is to start an episode with a random state-action pair.
  - This is called exploring starts.

2. For M number of iterations:
   1. **Select the initial state s_0 and initial action a_0 randomly such that all state-action pairs have a probability greater than 0**
   2. Generate an episode from the selected initial state s_0 and action a_0 using policy π
   3. Store all the rewards obtained in the episode in the list called rewards
   4. For each step t in the episode:

- Drawback of the exploring starts method
  - We do not know how to allocate probability for selecting an initial state-action pair.
  - Some states rarely occur while other states occur more frequently.
  - The exploring starts method does not reflect this.

On-Policy Monte Carlo Control

- Greedy policy
  - A greedy policy is one that selects the best action available at the moment.
  - If we have a Q table as below, the greedy policy would always choose action "up" in state A.

State	Action	Value
A	up	3
A	right	1

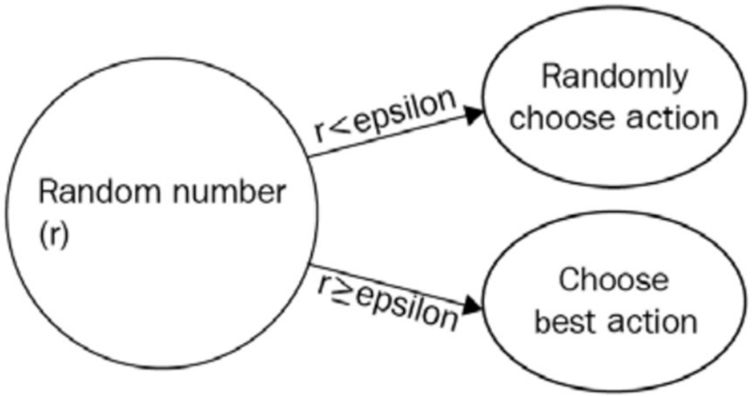
  - Suppose we did not explore actions "down" and "left" in state A, and their Q values are initialized to zeros.
  - Then, using a greedy policy, we might never be able to explore those actions, although they might turn out to be better than "up".

- Exploration-exploitation dilemma
  - Whether to choose the best action or to try other actions is called the exploration-exploitation dilemma.

On-Policy Monte Carlo Control

- Epsilon-greedy policy
  - A strategy to mix exploration and exploitation together.
  - In most cases, we choose the best action based on the Q table (exploitation).
  - But, with small probability ϵ, we choose a random action and explore new trajectories in the environment (exploration).

  - If $\epsilon = 0$, the policy becomes a greedy policy.
  - If $\epsilon = 1$, the policy becomes a policy which always chooses exploration.
  - The value of ϵ is typically chosen as a value between 0 and 1.



On-Policy Monte Carlo Control

- Python code that implements the epsilon-greedy policy

```
def epsilon_greedy_policy(state, epsilon):
    if random.uniform(0,1) < epsilon:
         return env.action_space.sample()
    else:
         return max(list(range(env.action_space.n)), key = lambda x:
q[(state,x)])
```

On-Policy Monte Carlo Control

- On-Policy Monte Carlo Control with Epsilon-Greedy Policy

2. For M number of iterations:
   1. Generate an episode using policy π
   2. Store all rewards obtained in the episode in the list called rewards
   3. For each step t in the episode:

      If (s_t, a_t) is occurring for the first time in the episode:
      1. Compute the return of a state-action pair, $R(s_t, a_t)$ = sum(rewards[t:])
      2. Update the total return of the state-action pair as total_return(s_t, a_t) = total_return(s_t, a_t) + R(s_t, a_t)
      3. Update the counter as $N(s_t, a_t) = N(s_t, a_t) + 1$
      4. Compute the Q value by just taking the average, that is,

$$Q(s_t, a_t) = \frac{\text{total\_return}(s_t, a_t)}{N(s_t, a_t)}$$

   4. Compute the updated policy π using the Q function. Let $a^* = \arg\max_a Q(s,a)$. The policy π selects the best action a^* with probability $1 - \epsilon$ and random action with probability ϵ

Implementing On-Policy Monte Carlo Control

- Create environment and prepare variables

```
import gym
import pandas as pd
import random
from collections import defaultdict
from tqdm import tqdm

env = gym.make('Blackjack-v0')
Q = defaultdict(float)
total_return = defaultdict(float)
N = defaultdict(int)
num_timesteps = 100
```

- Define the epsilon-greedy policy
  - epsilon is set to 0.5

```
def epsilon_greedy_policy(state, Q):
    epsilon = 0.5
    if random.uniform(0, 1) < epsilon:
        return env.action_space.sample()
    else:
        return max(list(range(env.action_space.n)), key = lambda x: Q[(state, x)])
```

Implementing On-Policy Monte Carlo Control

- Define a function that generates an episode based on the Q function

```
def generate_episode(Q):
    episode = []
    state = env.reset()
    for t in range(num_timesteps):
        action = epsilon_greedy_policy(state, Q)
        next_state, reward, done, info = env.step(action)
        episode.append((state, action, reward))
        if done:
            break
        state = next_state

    return episode
```

Implementing On-Policy Monte Carlo Control

- Run multiple episode, updating the Q function after each episode
  - Updating the Q function will change the policy → control task

```
num_iterations = 500000
for i in tqdm(range(num_iterations)):
    episode = generate_episode(Q)
    all_state_action_pairs = [(s,a) for (s,a,r) in episode]
    rewards = [r for (s,a,r) in episode]
    for t, (state, action, _) in enumerate(episode):
        if not (state, action) in all_state_action_pairs[0:t]:
            R = sum(rewards[t:])
            total_return[(state,action)] = total_return[(state,action)] + R
            N[(state,action)] += 1
            Q[(state,action)] = total_return[(state,action)] / N[(state,action)]
```

- The policy can be extracted from the Q function

```
df.sort_values(by=['state_action_pair'], inplace=True, ignore_index=True)
df[558:560]
```

✓ 0.6s

	state_action_pair	value
558	((21, 10, True), 0)	0.887300
559	((21, 10, True), 1)	-0.152707

Off-Policy Monte Carlo Control

- In the off-policy method, we use two policies
  - behavior policy b
  - target policy π

- The agent generates an episode using the behavior policy b.
- For each step in the episode, we compute the return of the state-action pair and compute the Q function $Q(s_t, a_t)$ as an average return.
- From this Q function, we extract a new target policy π.
- We repeat this step iteratively to find the optimal policy π.

- In off-policy method, the behavior policy b is not updated by the iteration.
- To explore various state-action pairs, the behavior policy b is set to an epsilon-greedy policy

Off-Policy Monte Carlo Control Algorithm

1. Initialize the Q function $Q(s,a)$ with random values, set the behavior policy b to be epsilon-greedy, and also set the target policy π to be a greedy policy.
2. For M number of episodes:
    1. Generate an episode using the behavior policy b
    2. Initialize return R to 0
    3. For each step t in the episode, t = T-1, T-2, ..., 0:
        1. Compute the return as $R = R + r_{t+1}$
        2. Compute the Q value as $Q(s_t, a_t) += \alpha(R_t - Q(s_t, a_t))$
        3. Compute the target policy $\pi(s_t) = arg \max_a Q(s_t, a)$
3. Return the target policy π

Off-Policy Monte Carlo Control Algorithm

- The problem of off-policy method
  - Difference of distribution between the behavior policy and the target policy can cause the target policy to be inaccurate.
  - To correct this, a technique called importance sampling is used.

- Importance Sampling
  - A technique for estimating the values of one distribution when given samples from another distribution

Importance Sampling

- The expectation of a function $f(x)$ where the value of x is sampled from the distribution $p(x)$ that is, $x \sim p(x)$; then we can write:

$$\mathbb{E}_{x \sim p(x)}[f(x)] = \int_x p(x) f(x) dx$$

- In the importance sampling method, we estimate the expectation using a different distribution $q(x)$.
- Instead of sampling x from $p(x)$ we use a different distribution $q(x)$ as shown as follows:

$$E_{x \sim p}[f(x)] = \int p(x) f(x) dx = \int \frac{p(x)}{q(x)} q(x) f(x) dx = E_{x \sim q}\left[\frac{p(x)}{q(x)} f(x) \right]$$

$$E_{x \sim p}[f(x)] \approx \frac{1}{N} \sum_{n=1}^{N} \frac{p(x_n)}{q(x_n)} f(x_n), \quad x_n \sim q$$

- Importance sampling ratio: $\dfrac{p(x)}{q(x)}$

Importance Sampling in Off-Policy Method

- Since we generate an episode to obtain return from a state, it will be inaccurate because the actions taken from the behavior policy b is different from the actions taken from the target policy π.

- We need to adjust the return using the importance sampling ratio.

- Given a starting state S_t, the probability of the subsequent state-action trajectory, $A_t, S_{t+1}, A_{t+1}, \dots, S_T$, occurring under a policy π is

$$
\begin{aligned}
&\Pr\{A_t, S_{t+1}, A_{t+1}, \dots, S_T \mid S_t, A_{t:T-1} \sim \pi\} \\
&\quad = \pi(A_t|S_t)p(S_{t+1}|S_t, A_t)\pi(A_{t+1}|S_{t+1}) \cdots p(S_T|S_{T-1}, A_{T-1}) \\
&\quad = \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k),
\end{aligned}
$$

  - where p is the transition probability.

Importance Sampling in Off-Policy Method

- The relative probability of the trajectory under the target and behavior policies (the importance sampling ratio) is:

$$\rho_{t:T-1} \doteq \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

- If we get the expected return starting from state s using the behavior policy b, it will be the value of state s under policy b.

$$\mathbb{E}[G_t|S_t = s] = v_b(s)$$

- In order to get $v_\pi(s)$, we need to use importance sampling ratio to transform the return.

$$\mathbb{E}[\rho_{t:T-1}G_t \mid S_t = s] = v_\pi(s)$$

Importance Sampling in Off-Policy Method

- Suppose we concatenate all the episodes.
- For example, if the first episode ends at time 100, then the next episode starts at time 101.

- Let $\mathcal{T}(s)$ denote the set of all time steps in which state s is visited.
- Also, let $T(t)$ denote the first time of termination following time t, and $G_t$denote the return after t up through $T(t)$.
- Then, $\{G_t\}_{t\in\mathcal{T}(s)}$ are the returns that pertain to state s, and
- $\{\rho_{t:T(t)-1}\}_{t\in\mathcal{T}(s)}$ are the corresponding importance-sampling ratios.

- To estimate $v_\pi(s)$, we scale the returns by the ratios and average the results.

$$V(s) \doteq \frac{\sum_{t\in\mathfrak{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathfrak{T}(s)|}.$$

  – This is called ordinary importance sampling.

Importance Sampling in Off-Policy Method

- An alternative is weighted importance sampling, where we use a weighted average for calculating $V(s)$.

$$V(s) \doteq \frac{\sum_{t \in \mathfrak{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathfrak{T}(s)} \rho_{t:T(t)-1}}$$

- Characteristics of ordinary importance sampling and weighted importance sampling
  - Ordinary importance sampling is unbiased, whereas weighted importance sampling is biased.
  - The variance of ordinary importance sampling is unbounded whereas the variance of weighted importance sampling converges to zero.

Weighted Importance Sampling using Incremental Mean

- Suppose we have a sequence of returns $G_1, G_2, \ldots, G_{n-1}$ all starting in the same state.
- Each return has a corresponding weight W_i ($W_i = \rho_{t_i:T(t_i)-1}$).
- We wish to calculate $V(n)$ using the previous returns we observed.

$$V_n \doteq \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}, \qquad n \geq 2.$$

- Once we obtain an additional return G_n, we update V_n using incremental mean. The update rule for V_n is:

$$V_{n+1} \doteq V_n + \frac{W_n}{C_n}\Big[G_n - V_n\Big], \qquad n \geq 1$$

- and

$$C_{n+1} \doteq C_n + W_{n+1}$$

Off-Policy Control with Weighted Importance Sampling

- We generate an episode using the behavior policy.
- We initialize return R to 0 and the weight W to 1.
- On every step of the episode in reverse order, we compute the return and update the cumulative weight as $C(s_t, a_t) = C(s_t, a_t) + W$.

- After updating the cumulative weight, we update the Q value as
  - $Q(s_t, a_t) = Q(s_t, a_t) + \frac{W}{C(s_t, a_t)}(R_t - Q(s_t, a_t))$

- From the Q value, we extract the target policy $\pi(s_t) = arg\max_a Q(s_t, a)$

- When the action a_t given by the behavior policy and the target policy is not the same, then we break the loop and generate the next episode.
- Else, we update the weight as: $W = W\frac{1}{b(a_t|s_t)}$

Off-Policy Control with Weighted Importance Sampling

- The algorithm

1. Initialize the Q function $Q(s, a)$ with random values, set the behavior policy b to be epsilon-greedy, and target policy π to be greedy policy and initialize the cumulative weights as $C(s, a) = 0$
2. For M number of episodes:
    1. Generate an episode using the behavior policy b
    2. Initialize return R to 0 and weight W to 1
    3. For each step t in the episode, $t = T\text{-}1, T\text{-}2, \ldots, 0$:
        1. Compute the return as $R = R + r_{t+1}$
        2. Update the cumulative weights $C(s_t, a_t) = C(s_t, a_t) + W$
        3. Update the Q value as $$Q(s_t, a_t) = Q(s_t, a_t) + \frac{W}{C(s_t, a_t)}\left(R_t - Q(s_t, a_t)\right)$$
        4. Compute the target policy $\pi(s_t) = \arg\max_a Q(s_t, a)$
        5. If $a_t \neq \pi(s_t)$ then break
        6. Update the weight as $W = W \dfrac{1}{b(a_t|s_t)}$
3. Return the target policy π

Off-Policy Control with Weighted Importance Sampling [ex006]

- Imports and environment setup

```
import sys
import gym
import numpy as np
from collections import defaultdict
env = gym.make('Blackjack-v0')
```

- Function that returns the random policy

```
def random_policy(nA):
    A = np.ones(nA, dtype=float) / nA
    def policy_fn(observation):
        return A
    return policy_fn
```

- Function that returns the greedy policy based on the Q function

```
def greedy_policy(Q):
    def policy_fn(state):
        A = np.zeros_like(Q[state], dtype=float)
        best_action = np.argmax(Q[state])
        A[best_action] = 1.0
        return A
    return policy_fn
```

Off-Policy Control with Weighted Importance Sampling [ex006]

- Implementation of the of-policy MC control (1/2)

```
def mc_off_policy(env, num_episodes, behavior_policy, max_time=100, discount_factor=1.0):
    Q = defaultdict(lambda:np.zeros(env.action_space.n))
    C = defaultdict(lambda:np.zeros(env.action_space.n))

    target_policy = greedy_policy(Q)

    for i_episode in range(1, num_episodes+1):
        if i_episode % 1000 == 0:
            print("\rEpisode {}/{}.".format(i_episode, num_episodes), end="")
            sys.stdout.flush()

        episode = []
        state = env.reset()
        for t in range(max_time):
            probs = behavior_policy(state)
            action = np.random.choice(np.arange(len(probs)), p=probs)
            next_state, reward, done, _ = env.step(action)
            episode.append((state, action, reward))
            if done:
                break
            state = next_state
```

Off-Policy Control with Weighted Importance Sampling [ex006]

- Implementation of the of-policy MC control (2/2)

```
        G = 0.0
        W = 1.0
        for t in range(len(episode))[::-1]:
            state, action, reward = episode[t]
            G = discount_factor * G + reward
            C[state][action] += W
            Q[state][action] += (W / C[state][action]) * (G - Q[state][action])
            if action != np.argmax(target_policy(state)):
                break
            W = W * 1./behavior_policy(state)[action]

    return Q, target_policy
```

- Calling functions to obtain the optimal policy

```
random_policy = random_policy(env.action_space.n)
Q, policy = mc_off_policy(env, num_episodes=500000, behavior_policy=random_policy)
```

Is the Monte Carlo method applicable to all tasks?

- Since the Monte Carlo is a model-free learning method, it can be applied to environments where the transition probability is not known (or too complex).

- The Monte Carlo method is only applicable to episodic tasks, because it relies on averaging sample returns.
  - We need to run multiple episodes to compute value or Q functions, or update the policy.

- It cannot be applied to continuous tasks.

- For continuous tasks, we have another model-free method called temporal difference learning.

End of Chapter

- Can you use the Monte Carlo control method to train an agent to play the Blackjack game?

- Can you write [ex004], [ex005], and [ex006] yourself?

End of Class

Questions?

Email: jso1@sogang.ac.kr