Gaming with Monte Carlo Methods

Monte Carlo is one of the most popular and most commonly used algorithms in various fields ranging from physics and mechanics to computer science. The Monte Carlo algorithm is used in **reinforcement learning** (**RL**) when the model of the environment is not known. In the previous chapter, `Chapter 3`, *Markov Decision Process and Dynamic Programming*, we looked at using **dynamic programming** (**DP**) to find an optimal policy where we know the model dynamics, which is transition and reward probabilities. But how can we determine the optimal policy when we don't know the model dynamics? In that case, we use the Monte Carlo algorithm; it is extremely powerful for finding optimal policies when we don't have knowledge of the environment.

In this chapter, you will learn about the following:

- Monte Carlo methods
- Monte Carlo prediction
- Playing Blackjack with Monte Carlo
- Model Carlo control
- Monte Carlo exploration starts
- On-policy Monte Carlo control
- Off-policy Monte Carlo control

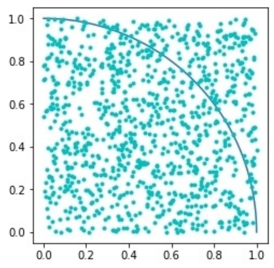
Monte Carlo methods

The Monte Carlo method finds approximate solutions through random sampling, that is, it approximates the probability of an outcome by running multiple trails. It is a statistical technique to find an approximate answer through sampling. Let's better understand Monte Carlo intuitively with an example.

Fun fact: *Monte Carlo is named after Stanislaw Ulam's uncle, who often borrowed money from his relatives to gamble in a Monte Carlo casino.*

Estimating the value of pi using Monte Carlo

Imagine a quadrant of a circle is placed inside a square, as shown next, and we generate some random points inside the square. You can see that some of the points fall inside the circle while others are outside the circle:



We can write:

$$\frac{Area \quad of \quad a \quad cirlce}{Area \quad of \quad a \quad square} = \frac{No \quad of \quad points \quad inside \quad the \quad circle}{No \quad of \quad points \quad inside \quad the \quad square}$$

We know that the area of a circle is πr^2 and the area of a square is a^2:

$$\frac{\pi r^2}{a^2} = \frac{No \quad of \quad points \quad inside \quad the \quad circle}{No \quad of \quad points \quad inside \quad the \quad square}$$

Let's consider that the radius of a circle is one half and the square's side is *1*, so we can substitute:

$$\frac{\pi(\frac{1}{2})^2}{1^2} = \frac{No \quad of \quad points \quad inside \quad the \quad circle}{No \quad of \quad points \quad inside \quad the \quad square}$$

Now we get the following:

$$\pi = 4 * \frac{No \quad of \quad points \quad inside \quad the \quad circle}{No \quad of \quad points \quad inside \quad the \quad square}$$

The steps to estimate π are very simple:

1. First, we generate some random points inside the square.

2. Then we can calculate the number of points that fall inside the circle by using the equation $x^2 + y^2 <= size$.
3. Then we calculate the value of π by multiplying four to the division of the number of points inside the circle to the number of points inside the square.
4. If we increase the number of samples (number of random points), the better we can approximate

Let's see how to do this in Python step by step. First, we import necessary libraries:

```
import numpy as np
import math
import random
import matplotlib.pyplot as plt
%matplotlib inline
```

Now we initialize the square size and number of points inside the circle and square. We also initialize the sample size, which denotes the number of random points to be generated. We define `arc`, which is basically the circle quadrant:

```
square_size = 1
points_inside_circle = 0
points_inside_square = 0
sample_size = 1000
arc = np.linspace(0, np.pi/2, 100)
```

Then we define a function called `generate_points()`, which generates random points inside the square:

```
def generate_points(size):
    x = random.random()*size
    y = random.random()*size
    return (x, y)
```

We define a function called `is_in_circle()`, which will check if the point we generated falls within the circle:

```
def is_in_circle(point, size):
    return math.sqrt(point[0]**2 + point[1]**2) <= size
```

Then we define a function for calculating the π value:

```
def compute_pi(points_inside_circle, points_inside_square):
    return 4 * (points_inside_circle / points_inside_square)
```

Then for the number of samples, we generate some random points inside the square and increment our `points_inside_square` variable, and then we will check if the points we generated lie inside the circle. If yes, then we increment the `points_inside_circle` variable:

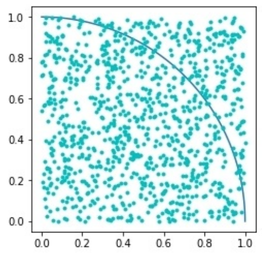
```
plt.axes().set_aspect('equal')
plt.plot(1*np.cos(arc), 1*np.sin(arc))

for i in range(sample_size):
    point = generate_points(square_size)
    plt.plot(point[0], point[1], 'c.')
    points_inside_square += 1
    if is_in_circle(point, square_size):
        points_inside_circle += 1
```

Now we calculate the value of π using the `compute_pi()`, function which will print an approximate value of π:

```
print("Approximate value of pi is {}" .format(calculate_pi(points_inside_circle,
points_inside_square)))
```

If you run the program, you will get the output shown as follows:

```
Approximate value of pi is 3.144
```



The complete program looks as follows:

```
import numpy as np
import math
import random
import matplotlib.pyplot as plt
%matplotlib inline

square_size = 1
points_inside_circle = 0
points_inside_square = 0
sample_size = 1000
arc = np.linspace(0, np.pi/2, 100)

def generate_points(size):
    x = random.random()*size
    y = random.random()*size
    return (x, y)

def is_in_circle(point, size):
    return math.sqrt(point[0]**2 + point[1]**2) <= size

def compute_pi(points_inside_circle, points_inside_square):
    return 4 * (points_inside_circle / points_inside_square)

plt.axes().set_aspect('equal')
plt.plot(1*np.cos(arc), 1*np.sin(arc))

for i in range(sample_size):
    point = generate_points(square_size)
    plt.plot(point[0], point[1], 'c.')
    points_inside_square += 1
    if is_in_circle(point, square_size):
        points_inside_circle += 1

print("Approximate value of pi is {}" .format(calculate_pi(points_inside_circle,
points_inside_square)))
```

Thus, the Monte Carlo method approximated the value of `pi` by using random sampling. We estimated the value of `pi` using the random points (samples) generated inside the square. The greater the sampling size, the better our approximation will be. Now we will see how to use Monte Carlo methods in RL.

Monte Carlo prediction

In DP, we solve the **Markov Decision Process** (**MDP**) by using value iteration and policy iteration. Both of these techniques require transition and reward probabilities to find the optimal policy. But how can we solve MDP when we don't know the transition and reward probabilities? In that case, we use the Monte Carlo method. The Monte Carlo method requires only sample sequences of states, actions, and rewards. the Monte Carlo methods are applied only to the episodic tasks. Since Monte Carlo doesn't require any model, it is called the model-free learning algorithm.

The basic idea of the Monte Carlo method is very simple. Do you recall how we defined the optimal value function and how we derived the optimal policy in the previous chapter, `Chapter 3`, *Markov Decision Process and Dynamic Programming*?

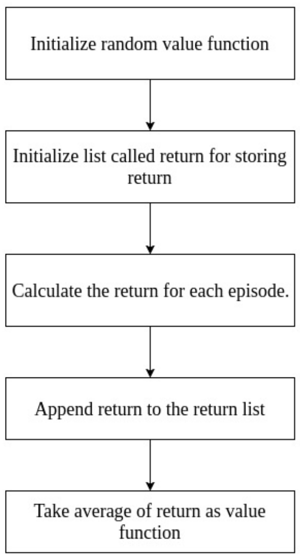
A value function is basically the expected return from a state S with a policy π. Here, instead of expected return, we use mean return.

Thus, in Monte Carlo prediction, we approximate the value function by taking the mean return instead of the expected return.

Using Monte Carlo prediction, we can estimate the value function of any given policy. The steps involved in the Monte Carlo prediction are very simple and are as follows:

1. First, we initialize a random value to our value function
2. Then we initialize an empty list called a return to store our returns
3. Then for each state in the episode, we calculate the return
4. Next, we append the return to our return list
5. Finally, we take the average of return as our value function

The following flowchart makes it more simple:


Initialize random value function
Initialize list called return for storing return
Calculate the return for each episode.
Append return to the return list
Take average of return as value function

The Monte Carlo prediction algorithm is of two types:

- First visit Monte Carlo
- Every visit Monte Carlo

First visit Monte Carlo

As we have seen, in the Monte Carlo methods, we approximate the value function by taking the average return. But in the first visit MC method, we average the return only the first time the state is visited in an episode. For example, consider an agent is playing the snakes and ladder games, there is a good chance the agent will return to the state if it is bitten by a snake. When the agent revisits the state, we don't consider an average return. We consider an average return only when the agent visits the state for the first time.

Every visit Monte Carlo

In every visit Monte Carlo, we average the return every time the state is visited in an episode. Consider the same snakes and ladders game example: if the agent returns to the same state after a snake bites it, we can think of this as an average return although the agent is revisiting the state. In this case, we average return every time the agents visit the state.

Let's play Blackjack with Monte Carlo

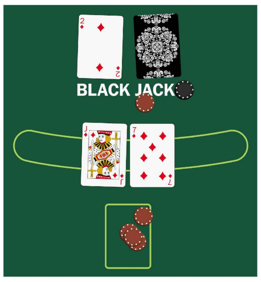
Now let's better understand Monte Carlo with the Blackjack game. Blackjack, also called 21, is a popular card game played in casinos. The goal of the game is to have a sum of all your cards close to 21 and not exceeding 21. The value of cards J, K, and Q is 10. The value of ace can be 1 or 11; this depends on player choice. The value of the rest of the cards (1 to 10) is the same as the numbers they show.

The rules of the game are very simple:

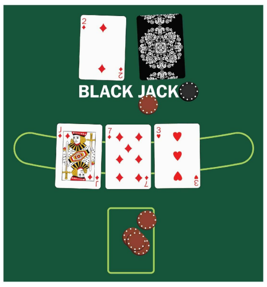
- The game can be played with one or many players and one dealer.
- Each player competes only with the dealer and not another player.
- Initially, a player is given two cards. Both of these cards are face up, that is, visible to others.
- A dealer is also given two cards. One card is face up and the other is face down. That is, the dealer only shows one of his cards.

- If the sum of a player's cards is 21 immediately after receiving two cards (say a player has received a jack and ace which is 10+11 = 21), then it is called **natural** or **Blackjack** and the player wins.
- If the dealer's sum of cards is also 21 immediately after receiving two cards, then it is called a **draw** as both of them have 21.
- In each round, the player decides whether he needs another card or not to sum the cards close to 21.
- If a player needs a card, then it is called a **hit**.
- If a player doesn't need a card, then it is called a **stand**.
- If a player's sum of cards exceeds 21, then it is called **bust**; then the dealer will win the game.

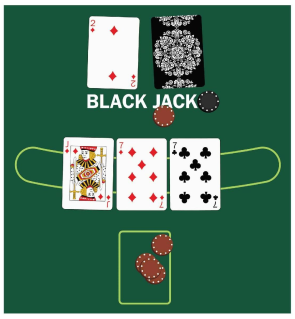
Let's better understand Blackjack by playing. I'll let you be the player and I am the dealer:



In the preceding diagram, we have one player and a dealer. Both of them are given two cards. Both of the player's two cards are face up (visible) while the dealer has one card face up (visible) and the other face down (invisible). In the first round, you have been given two cards, say a jack and a number 7, which is (10 + 7 = 17), and I as the dealer will only show you one card which is number 2. I have another card face down. Now you have to decide to either hit (need another card) or stand (don't need another card). If you choose to hit and receive number 3 you will get 10+7+3 = 20 which is close to 21 and you win:



But if you received a card, say number 7, then 10+7+7 = 24, which exceeds 21. Then it is called bust and you lose the game. If you decide to stand with your initial cards, then you have only 10 + 7 = 17. Then we will check the dealer's sum of cards. If it is greater than 17 and does not exceed 21 then the dealer wins, otherwise you win:



The rewards here are:

- +1 if the player won the game
- -1 if the player loses the game
- 0 if the game is a draw

The possible actions are:

- **Hit**: If the player needs a card
- **Stand**: If the player doesn't need a card

The player has to decide the value of an ace. If the player's sum of cards is 10 and the player gets an ace after a hit, he can consider it as 11, and 10 + 11 = 21. But if the player's sum of cards is 15 and the player gets an ace after a hit, if he considers it as 11 and 15+11 = 26, then it's a bust. If the player has an ace we can call it a **usable ace**; the player can consider it as 11 without being bust. If the player is bust by considering the ace as 11, then it is called a **nonusable ace**.

Now we will see how to implement Blackjack using the first visit Monte Carlo algorithm.

First, we will import our necessary libraries:

```
import gym
from matplotlib import pyplot
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from collections import defaultdict
from functools import partial
%matplotlib inline
plt.style.use('ggplot')
```

Now we will create the Blackjack environment using OpenAI's Gym:

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

Then we define the policy function which takes the current state and checks if the score is greater than or equal to `20`; if it is, we return `0` or else we return `1`. That is, if the score is greater than or equal to `20`, we stand (0) or else we hit (1):

```
def sample_policy(observation):
    score, dealer_score, usable_ace = observation
    return 0 if score >= 20 else 1
```

Now we will see how to generate an episode. An episode is a single round of a game. We will see it step by step and then look at the complete function.

We define states, actions, and rewards as a list and initiate the environment using `env.reset` and store an observation variable:

```
states, actions, rewards = [], [], []
observation = env.reset()
```

Then, until we reach the terminal state, that is, till the end of the episode, we do the following:

1. Append the observation to the states list:

```
        states.append(observation)
```

2. Now, we create an action using our `sample_policy` function and append the actions to an `action` list:

```
        action = sample_policy(observation)
        actions.append(action)
```

3. Then, for each step in the environment, we store the `state`, `reward`, and `done` (which specifies whether we reached terminal state) and we append the rewards to the `reward` list:

```
        observation, reward, done, info = env.step(action)
        rewards.append(reward)
```

4. If we reached the terminal state, then we break:

```
        if done:
            break
```

5. The complete `generate_episode` function is as follows:

```
        def generate_episode(policy, env):
            states, actions, rewards = [], [], []
            observation = env.reset()
            while True:
                states.append(observation)
                action = policy(observation)
                actions.append(action)
                observation, reward, done, info = env.step(action)
                rewards.append(reward)
                if done:
                    break

            return states, actions, rewards
```

This is how we generate an episode. How can we play the game? For that, we need to know the value of each state. Now we will see how to get the value of each state using the first visit Monte Carlo method.

First, we initialize the empty value table as a dictionary for storing the values of each state:

```
value_table = defaultdict(float)
```

Then, for a certain number of episodes, we do the following:

1. First, we generate an episode and store the states and rewards; we initialize returns as `0` which is the sum of rewards:

```
        states, _, rewards = generate_episode(policy, env)
        returns = 0
```

2. Then for each step, we store the rewards to a variable *R* and states to *S*, and we calculate returns as a sum of rewards:

```
        for t in range(len(states) - 1, -1, -1):
            R = rewards[t]
            S = states[t]
            returns += R
```

3. We now perform the first visit Monte Carlo; we check if the episode is being visited for the visit time. If it is, we simply take the average of returns and assign the value of the state as an average of returns:

```
        if S not in states[:t]:
            N[S] += 1
            value_table[S] += (returns - V[S]) / N[S]
```

4. Look at the complete function for better understanding:

```
        def first_visit_mc_prediction(policy, env, n_episodes):
            value_table = defaultdict(float)
            N = defaultdict(int)

            for _ in range(n_episodes):
                states, _, rewards = generate_episode(policy, env)
                returns = 0
                for t in range(len(states) - 1, -1, -1):
                    R = rewards[t]
                    S = states[t]
                    returns += R
                    if S not in states[:t]:
                        N[S] += 1
                        value_table[S] += (returns - V[S]) / N[S]
            return value_table
```

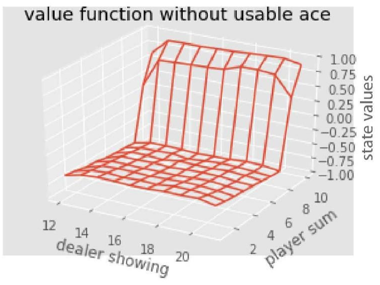
5. We can get the value of each state:

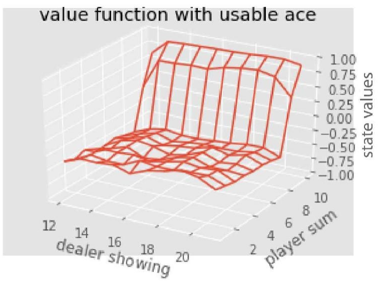
```
value = first_visit_mc_prediction(sample_policy, env, n_episodes=500000)
```

6. Let's see the value of a few states:

```
print(value)
defaultdict(float,
            {(4, 1, False): -1.024292170184644,
             (4, 2, False): -1.8670191351012455,
             (4, 3, False): 2.211363314854649,
             (4, 4, False): 16.903201033000823,
             (4, 5, False): -5.786238030898542,
             (4, 6, False): -16.218211752577602,
```

We can also plot the value of the state to see how it is converged, as follows:





The complete code is given as follows:

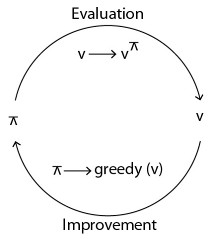
```
import numpy
import gym
from matplotlib import pyplot
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from collections import defaultdict
from functools import partial
%matplotlib inline

plt.style.use('ggplot')

## Blackjack Environment

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

```
env.action_space, env.observation_space

def sample_policy(observation):
    score, dealer_score, usable_ace = observation
    return 0 if score >= 20 else 1

def generate_episode(policy, env):
    states, actions, rewards = [], [], []
    observation = env.reset()
    while True:
        states.append(observation)
        action = sample_policy(observation)
        actions.append(action)
        observation, reward, done, info = env.step(action)
        rewards.append(reward)
        if done:
            break

    return states, actions, rewards


def first_visit_mc_prediction(policy, env, n_episodes):
    value_table = defaultdict(float)
    N = defaultdict(int)

    for _ in range(n_episodes):
        states, _, rewards = generate_episode(policy, env)
        returns = 0
        for t in range(len(states) - 1, -1, -1):
            R = rewards[t]
            S = states[t]
            returns += R
            if S not in states[:t]:
                N[S] += 1
                value_table[S] += (returns - value_table[S]) / N[S]
    return value_table

def plot_blackjack(V, ax1, ax2):
    player_sum = numpy.arange(12, 21 + 1)
    dealer_show = numpy.arange(1, 10 + 1)
    usable_ace = numpy.array([False, True])

    state_values = numpy.zeros((len(player_sum),
                                len(dealer_show),
                                len(usable_ace)))

    for i, player in enumerate(player_sum):
        for j, dealer in enumerate(dealer_show):
            for k, ace in enumerate(usable_ace):
                state_values[i, j, k] = V[player, dealer, ace]

    X, Y = numpy.meshgrid(player_sum, dealer_show)

    ax1.plot_wireframe(X, Y, state_values[:, :, 0])
    ax2.plot_wireframe(X, Y, state_values[:, :, 1])
    for ax in ax1, ax2:
        ax.set_zlim(-1, 1)
        ax.set_ylabel('player sum')
        ax.set_xlabel('dealer showing')
        ax.set_zlabel('state-value')

fig, axes = pyplot.subplots(nrows=2, figsize=(5, 8), subplot_kw={'projection': '3d'})
axes[0].set_title('value function without usable ace')
axes[1].set_title('value function with usable ace')
```

```
plot_blackjack(value, axes[0], axes[1])
```

Monte Carlo control

In Monte Carlo prediction, we have seen how to estimate the value function. In Monte Carlo control, we will see how to optimize the value function, that is, how to make the value function more accurate than the estimation. In the control methods, we follow a new type of iteration called generalized policy iteration, where policy evaluation and policy improvement interact with each other. It basically runs as a loop between policy evaluation and improvement, that is, the policy is always improved with respect to the value function, and the value function is always improved according to the policy. It keeps on doing this. When there is no change, then we can say that the policy and value function have attained convergence, that is, we found the optimal value function and optimal policy:


Evaluation
$v \longrightarrow v^{\pi}$
π
v
$\pi \longrightarrow \text{greedy}(v)$
Improvement

Now we will see a different Monte Carlo control algorithm as follows.

Monte Carlo exploration starts

Unlike DP methods, here we do not estimate state values. Instead, we focus on action values. State values alone are sufficient when we know the model of the environment. As we don't know about the model dynamics, it is not a good way to determine the state values alone.

Estimating an action value is more intuitive than estimating a state value because state values vary depending on the policy we choose. For example, in a Blackjack game, say we are in a state where some of the cards are 20. What is the value of this state? It solely depends on the policy. If we choose our policy as a hit, then it is not a good state to be in and the value of this state is very low. However, if we choose our policy as a stand then it is definitely a good state to be in. Thus, the value of the state depends on the policy we choose. So it is more important to estimate the value of an action instead of the value of the state.

How do we estimate the action values? Remember the *Q* function we learned in `Chapter 3`, *Markov Decision Process and Dynamic Programming*? The *Q* function denoted as *Q(s, a)* is used for determining how good an action is in a particular state. It basically specifies the state-action pair.

But here the problem of exploration comes in. How can we know about the state-action value if we haven't been in that state? If we don't explore all the states with all possible actions, we might probably miss out the good rewards.

Say that in a Blackjack game, we are in a state where a sum of cards is 20. If we try only the action **hit** we will get a negative reward, and we learn that it is not a good state to be in. But if we try the **stand** action, we receive a positive reward and it is actually the best state to be in. So every time we come to this particular state, we stand instead of hit. For us to know which is the best action, we have to explore all possible actions in each state to find the optimal value. How can we do this?

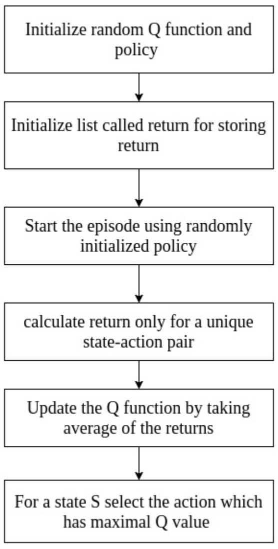
Let me introduce a new concept called **Monte Carlo exploring starts**, which implies that for each episode we start with a random state as an initial state and perform an action. So, if we have a large number of episodes, we could possibly cover all the states with all possible actions. It is also called an **MC-ES** algorithm.

The MC-ES algorithm is very simple, as follows:

- We first initialize *Q* function and policy with some random values and also we initialize a return to an empty list
- Then we start the episode with our randomly initialized policy
- Then we calculate the return for all the unique state-action pairs occurring in the episode and append return to our return list
- We calculate a return only for a unique state-action pair because the same state action pair occurs in an episode multiple times and there is no point having redundant information
- Then we take an average of the returns in the return list and assign that value to our *Q* function

- Finally, we will select an optimal policy for a state, choosing an action that has the maximum *Q(s,a)* for that state
- We repeat this whole process forever or for a large number of episodes so that we can cover all different states and action pairs

Here's a flowchart of this:


Initialize random Q function and policy
Initialize list called return for storing return
Start the episode using randomly initialized policy
calculate return only for a unique state-action pair
Update the Q function by taking average of the returns
For a state S select the action which has maximal Q value

On-policy Monte Carlo control

In Monte Carlo exploration starts, we explore all state-action pairs and choose the one that gives us the maximum value. But think of a situation where we have a large number of states and actions. In that case, if we use the MC-ES algorithm, then it will take a lot of time to explore all combinations of states and actions and to choose the best one. How do we get over this? There are two different control algorithms. On policy and off policy. In on-policy Monte Carlo control, we use the ε greedy policy. Let's understand what a greedy algorithm is.

A greedy algorithm picks up the best choice available at that moment, although that choice might not be optimal when you consider the overall problem. Consider you want to find the smallest number from a list of numbers. Instead of finding the smallest number directly from the list, you will divide the list into three sublists. Then you will find the smallest number in each of the sublists (local optima). The smallest number you find in one sublist might not be the smallest number when you consider the whole list (global optima). However, if you are acting greedy then you will see the smallest number in only the current sublist (at the moment) and consider it the smallest number.

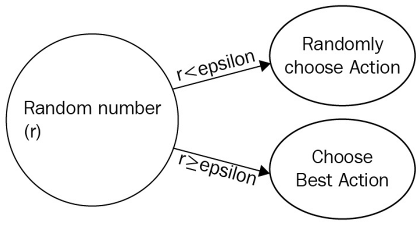
The greedy policy denotes the optimal action within the actions explored. The optimal action is the one which has the highest value.

Say we have explored some actions in the state 1, as shown in the Q table:

State	**Action**	**Value**
State 1	Action 0	0.5
State 1	Action 1	0.1
State 1	Action 2	0.8

If we are acting greedy, we would pick up the action that has maximal value out of all the actions we explored. In the preceding case, we have action 2 which has high value, so we pick up that action. But there might be other actions in the state 1 that we haven't explored and might the highest value. So we have to look for the best action or exploit the action that is best out of all explored actions. This is called an exploration-exploitation dilemma. Say you listened to Ed Sheeran and you liked him very much, so you kept on listening to Ed Sheeran only (exploiting) because you liked the music. But if you tried listening to other artists you might like someone better than Ed Sheeran (exploration). This confusion as to whether you have to listen to only Ed Sheeran (exploitation) or try listening to different artists to see if you like them (exploration) is called an exploration-exploitation dilemma.

So to avoid this dilemma, we introduce a new policy called the epsilon-greedy policy. Here, all actions are tried with a non-zero probability (epsilon). With a probability epsilon, we explore different actions randomly and with a probability 1-epsilon we choose an action that has maximum value, that is, we don't do any exploration. So instead of just exploiting the best action all the time, with probability epsilon, we explore different actions randomly. If the value of the epsilon is set to zero, then we will not do any exploration. It is simply the greedy policy, and if the value of epsilon is set to one, then it will always do only exploration. The value of the epsilon will decay over time as we don't want to explore forever. So over time our policy exploits good actions:



Let us say we set the value of epsilon to *0.3*. In the following code, we generate a random value from the uniform distribution and if the value is less than epsilon value, that is, 0.3, then we select a random action (in this way, we search for a different action). If the random value from the uniform distribution is greater than 0.3, then we select the action that has the best value. So, in this way, we explore actions that we haven't seen before with the probability epsilon and select the best actions out of the explored actions with the probability 1-epsilon:

```
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)])
```

Let us imagine that we have explored further actions in the state 1 with the epsilon-greedy policy (although not all of the actions pair) and our Q table looks as follows:

State	**Action**	**Value**
State 1	Action 0	0.5
State 1	Action 1	0.1
State 1	Action 2	0.8
State 1	Action 4	0.93

In state 1, action 4 has a higher value than the action 2 we found previously. So with the epsilon-greedy policy, we look for different actions with the probability epsilon and exploit the best action with the probability 1-epsilon.

The steps involved in the on-policy Monte Carlo method are very simple:

1. First, we initialize a random policy and a random Q function.
2. Then we initialize a list called return for storing the returns.
3. We generate an episode using the random policy π.
4. We store the return of every state action pair occurring in the episode to the return list.

5. Then we take an average of the returns in the return list and assign that value to the *Q* function.
6. Now the probability of selecting an action *a* in the state *s* will be decided by epsilon.
7. If the probability is 1-epsilon we pick up the action which has the maximal *Q* value.
8. If the probability is epsilon, we explore for different actions.

Off-policy Monte Carlo control

Off-policy Monte Carlo is another interesting Monte Carlo control method. In this method, we have two policies: one is a behavior policy and another is a target policy. In the off-policy method, agents follow one policy but in the meantime, it tries to learn and improve a different policy. The policy an agent follows is called a behavior policy and the policy an agent tries to evaluate and improve is called a target policy. The behavior and target policy are totally unrelated. The behavior policy explores all possible states and actions and that is why a behavior policy is called a soft policy, whereas a target policy is said to be a greedy policy (it selects the policy which has the maximal value).

Our goal is to estimate the *Q* function for the target policy π, but our agents behave using a completely different policy called behavior policy μ. What can we do now? We can estimate the value of π by using the common episodes that took place in μ. How can we estimate the common episodes between these two policies? We use a new technique called importance sampling. It is a technique for estimating values from one distribution given samples from another.

Importance sampling is of two types:

- Ordinary importance sampling
- Weighted importance sampling

In ordinary importance sampling, we basically take the ratio of returns obtained by the behavior policy and target policy, whereas in weighted importance sampling we take the weighted average and *C* is the cumulative sum of weights.

Let us just see this step by step:

1. First, we initialize *Q(s,a)* to random values and *C(s,a)* to *0* and weight *w* as *1*.
2. Then we choose the target policy, which is a greedy policy. This means it will pick up the policy which has a maximum value from the *Q* table.

3. We select our behavior policy. A behavior policy is not greedy and it can

select any state-action pair.

4. Then we begin our episode and perform an action *a* in the state *s* according to our behavior policy and store the reward. We repeat this until the end of the episode.
5. Now, for each state in the episode, we do the following:
    1. We will calculate return *G*. We know that the return is the sum of discounted rewards: *G = discount_ factor * G + reward*.
    2. Then we update *C(s,a)* as *C(s,a) = C(s,a) + w*.
    3. We update *Q(s,a)*: $Q(s,a) = Q(s,a) + \frac{w}{C(s,a)} * (G - Q(s,a))$.
    4. We update the value of *w*: $w = w * \frac{1}{behaviourpolicy}$.

Summary

In this chapter, we learned about how the Monte Carlo method works and how can we use it to solve MDP when we don't know the model of the environment. We have looked at two different methods: one is Monte Carlo prediction, which is used for estimating the value function, and the other is Monte Carlo control, which is used for optimizing the value function.

We looked at two different methods in Monte Carlo prediction: first visit Monte Carlo prediction, where we average the return only the first time the state is visited in an episode, and the every visit Monte Carlo method, where we average the return every time the state is visited in an episode.

In terms of Monte Carlo control, we looked at different algorithms. We first encountered MC-ES control, which is used to cover all state-action pairs. We looked at on-policy MC control, which uses the epsilon-greedy policy, and off-policy MC control, which uses two policies at a time.

In the next chapter, `Chapter 5`, *Temporal Difference Learning* we will look at a different model-free learning algorithm.