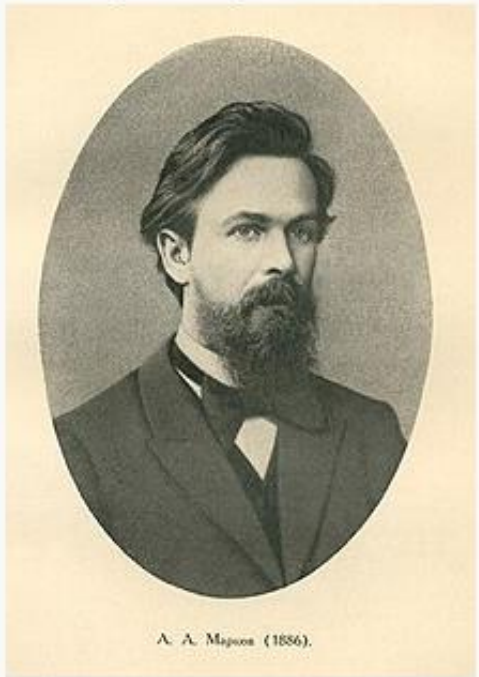
3.Monte Carlo Methods

Contents

- Review : So far, MDP and Dynamic Programing
- Monte Carlo (MC) method
- Monte Carlo(MC) in RL
- On-Policy MC
- Code Ex.
- Off-Policy MC
- Code Ex.

Review

Andrey Andreyevich Markov



Markov in 1886

Born	14 June 1856 N.S. Ryazan, Russian Empire
Died	20 July 1922 (aged 66) Petrograd, Russian SFSR
Nationality	Russian
Alma mater	St. Petersburg University
Known for	Markov chains Markov processes Stochastic processes

Richard Ernest Bellman

Born	Richard Ernest Bellman August 26, 1920 New York City, New York, U.S.
Died	March 19, 1984 (aged 63) Los Angeles, California, U.S.
Alma mater	Brooklyn College (BS) University of Wisconsin (MA) Princeton University (PhD)
Known for	Dynamic programming Stochastic dynamic programming Curse of dimensionality Linear search problem Bellman equation Bellman–Ford algorithm

Monte Carlo

Location in relation to Europe
Coordinates: 43°44′23″N 7°25′38″E

Country	Monaco
Established	1 June 1866
Government	
• **Type**	Principality
• **Prince of Monaco**	Albert II
Area	
• **Urban**	0.30 km^2 (.234 sq mi)
Population	
• **Quarter and ward**	15,200 (in the quarter) 3,500 (in the ward)
Postcode	98000

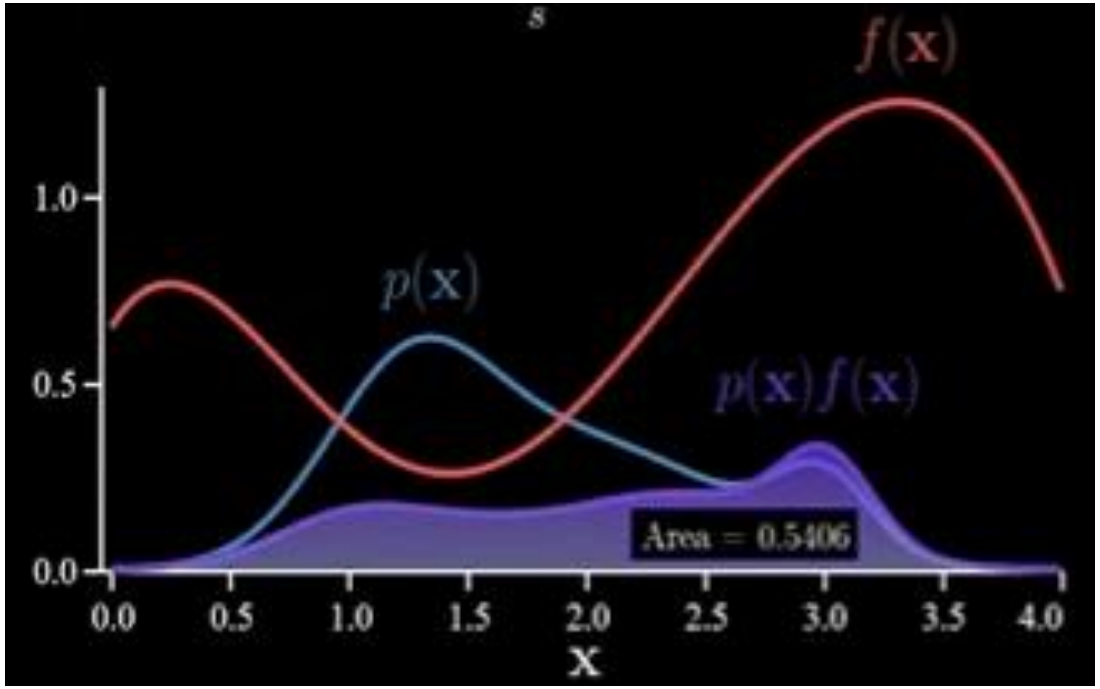
Understand MC estimation

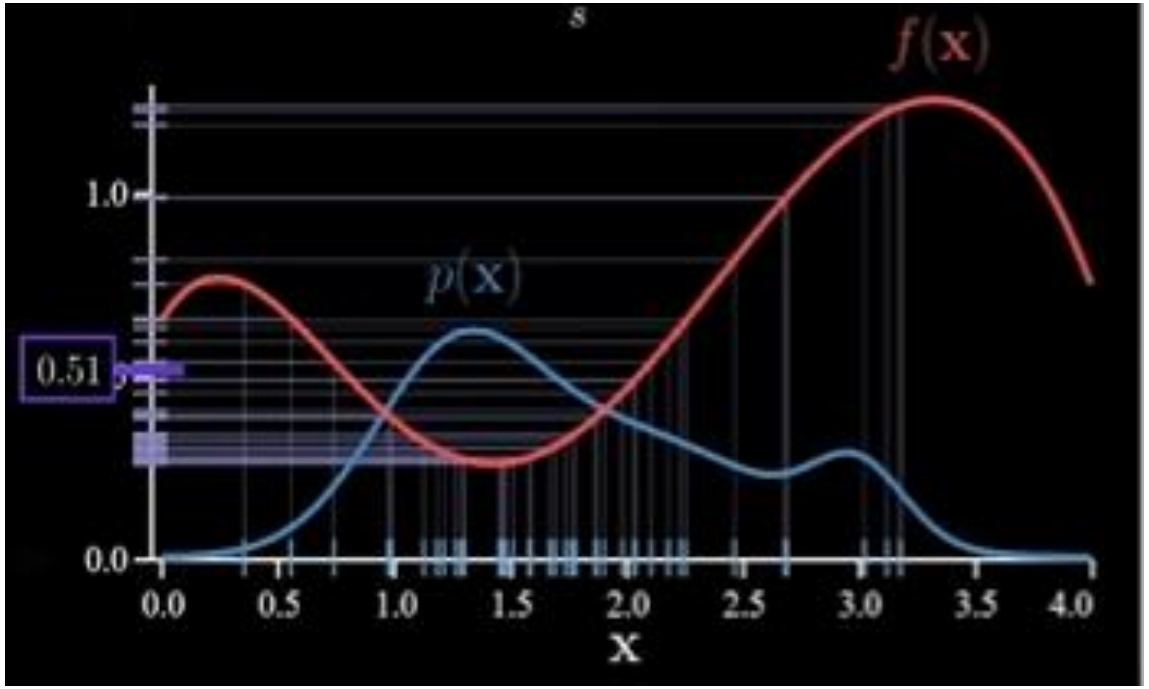
- Goal : calculate $\int p(x)f(x)dx$ = $E_p[f(x)]$
  - Prob. weighted average of f(x) over the entire space where x lives
  - If x is discrete case, $\sum_x p(x)f(x)$
  - Typically, x is high dim. So, the space that it lives within is exponentially huge, and impossible to add everything up within it

- Idea: approximate it with an average:

$$E_p[f(x)] \approx \frac{1}{N}\sum_{i=1}^{n} f(x_i) \qquad x_i \sim p(x)$$

Understand MC estimation

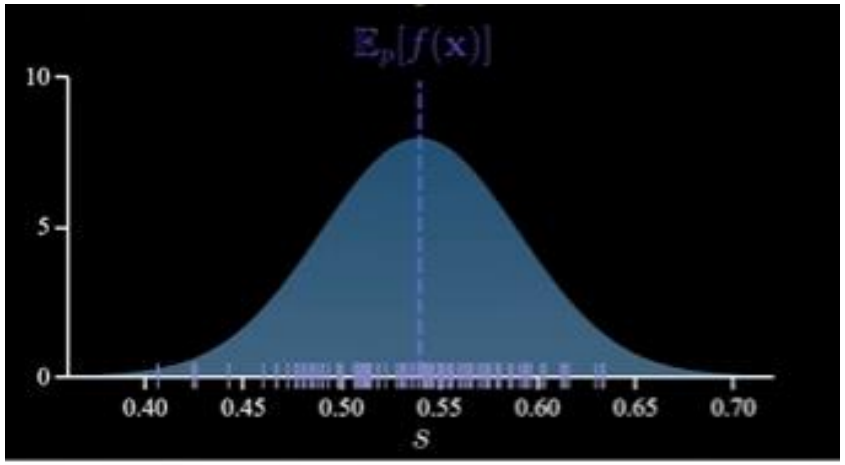




Understand MC estimation

- Unbiased Estimator
  - Note MC approximation(sample avg.) has a its own distribution depending on samples.
  - But it's centered on the true expectation we are after, $E_p[f(x)]$.

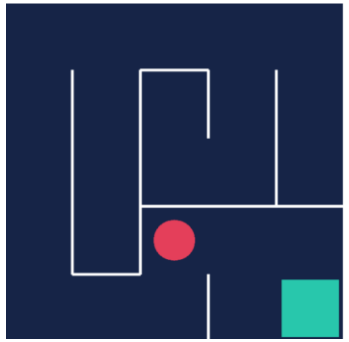
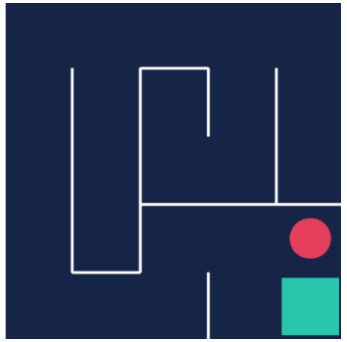
- By the Central Limit Theorem:
  - $\frac{1}{N}\sum_{i=1}^{n} f(x_i) \rightarrow N(\mu,\sigma^2)$ $\begin{cases} \mu = E_p[f(x)] \\ \sigma^2 = \frac{1}{N} V_p[f(x)] \end{cases}$



Monte Carlo in RL

MC method in RL

- Meet the first family of methods that learns through experience
- Family of methods that learn optimal v*(s) or Q*(s, a) values based on samples collected by the agent while interacting with the environment.
- The agent will use a policy π to tackle the task for an entire episode: $S_0, A_0, R_1, S_1, A_1, \dots, S_{T-1}, A_{T-1}, R_T$

MC method in RL

- At the end of the episode we'll compute the return from every state visited

- They approximate the values by interacting with the environment to generate sample returns and averaging them

$$v_\pi(s) = E_\pi[G_t \mid S_t = s], \qquad G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$$

$$V_\pi(s) = \frac{1}{N} \sum_{k=1}^{N} G_{sk}$$

MC method in RL

- They approximate the values by interacting with the environment to generate sample returns and averaging them

$$q_\pi(s, a) = E_\pi[G_t \mid S_t = s, A_t = a], \quad G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$$

$$Q_\pi(s, a) = \frac{1}{N} \sum_{k=1}^{N} G_{s, ak}$$

Law of large numbers

- In the limit, this succession of return samples, G_{s1}, G_{s2}, …, G_{sn} converges to its expected value $v_\pi(s)$

$$P(\lim_{n\to\infty} \widetilde{G}_s = v_\pi(s)) = 1$$

- The more experience, the more accurate our estimate

Advantages of MC

- MC estimate of a state value does not depend on the rest

- Dynamic Programming bootstraps the value of other states to estimate, meaning uses one estimate to produce another estimate.

$$v*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v*(s')]$$

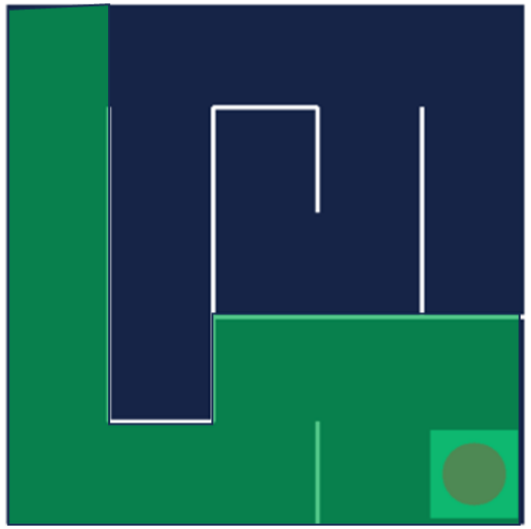
Advantages of MC

- The complexity of estimating the value of a state doesn't depend on the number of states in the task
  - The cost of estimating a state value is independent of the total number of state

- Dynamic Programming, the complexity of algorithms grows exponentially with the number of states

Advantages of MC

- MC can focus its efforts on estimating correctly the value of states that lead to the goal
- MC can focus the estimations on the states that help us solve the task



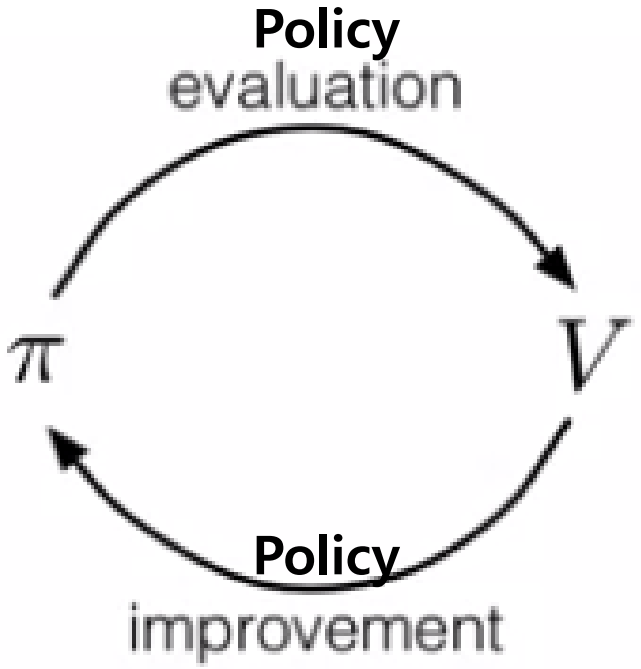
- The states shaded in green are more important than the rest because they form the optimal path and therefore we want to focus our learning on them
- Dynamic Programing sweeps through the state space and updates every single state whether they are important or not

Advantages of MC

- MC don’t require the model of env.
- No need to know the dynamics of the environment
- Dynamics will be implicit in our estimates
  - Note Dynamic Programing requires the model of environment
- For many tasks it is easier to generate samples than to model their dynamics

MC method in RL

- Remember Generalized Policy iteration results in the following iterative process (template) :



MC method in RL

- The agent will face the environment using the initial policy for one whole episode form start to finish

$$S_0, A_0, R_1, S_1, A_1, \dots, S_{T-1}, A_{T-1}, R_T$$

- From the generated trajectory, we will calculate the returns for each moment of time t :

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-t-1} R_T$$
$$G_{t+1} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-t-1} R_T$$

....

MC method in RL

- Our strategy is, to use those returns to evaluate the policy, and based on the value function, improve the policy π

- However, we can't use v(s) anymore to improve policy because v(s) requires knowing the effects of taking each action beforehand.

$$v*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v*(s')]$$

$$\pi'(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V_\pi(s')]$$

MC method in RL

- In MC, we don't have a model and no access to the dynamics

- Instead of keeping an estimate of the value of the states, we get an estimate of the expected return from taking each individual action in that state:

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$$

  - The environment dynamics are **implicit** in $q_\pi(s,a)$,
  - q estimate the value of taking an action in a state,
  - the value is the expected return from taking that action.

MC method in RL

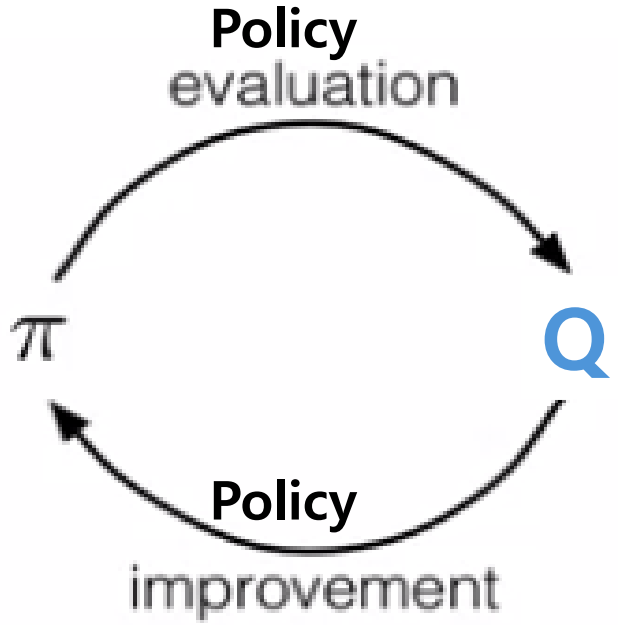
- Then it would be enough to compare those estimates and choose the action with the highest estimated return

- We'll change the policy to take the action with the highest $q_\pi(s, a)$:

$$\pi'(s) = \arg\max_a q_\pi(s, a)$$

- Instead of V(s), We keep a table with Q(s, a)

MC method in RL



$$\pi_0 \rightarrow Q_{\pi 0} \rightarrow \pi_1 \rightarrow Q_{\pi 1} \rightarrow \cdots \rightarrow Q_{\pi *} \rightarrow \pi *$$

Exploration

Exploration

- However, for this strategy to work, we have to keep something in mind :
  - We now improve our policy based on the experience that the agent collects while interacting with env.
  - The experience that the agent collects depends on the actions it takes
  - Those actions depend on the policy that the agent is using at that time

Q(s, a) is an estimate

$$\pi'(s) = \arg\max_{a} Q(s, a)$$

Exploration

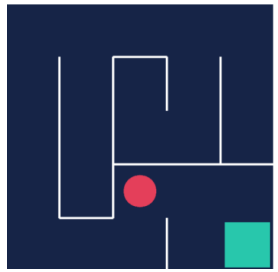
Q(s, a) is an estimate $\pi'(s) = \arg\max_a Q(s, a)$

- The estimates are poor especially in the early stage.
  - It will improve as we obtain new samples, but it might not be perfect
- If a is optimal but Q(s, a) is low, we'll never pick it.
  - The only way to correct this is to explore all actions every once in a while and update their estimate Q(s, a), so we do not leave a possible optimal action undiscovered

Exploration

- How to maintain Exploration:
  - Exploring starts:

$$S_0 \sim S,\ A_0 \sim A(S_0)$$



  - Stochastic policies:

$$\pi(a \mid s) > 0, \quad \forall a \in A(s)$$

This ensures that from time to time, it takes an action that it doesn't consider optimal to improve

Exploration

- Use of stochastic policies can be implemented in two different ways :
  - **On-policy** learning strategy generates samples using the same policy π that we're going to optimize
  - **Off-policy** learning strategy generates samples with an exploratory policy b different from the one (π) we're going go optimize

On-policy Monte Carlo

ϵ-greedy policy

- With probability ϵ , select a random action
  ,with probability 1- ϵ, select the action with the highest Q(s, a)

$$\pi(a \mid s) = \begin{cases} 1 - \epsilon + \epsilon_r & a = a* \\ \epsilon_r & a \neq a* \end{cases} \qquad \epsilon_r = \frac{\epsilon}{|A|}$$

- Example:

$|A| = 4, \quad \varepsilon = 0.2$

$$\pi(a \mid s) = \begin{cases} 1 - 0.2 + 0.05 = 0.85 & a = a* \\ 0.05 & a \neq a* \end{cases} \qquad \epsilon_r = \frac{0.2}{4} = 0.05$$

ϵ-greedy policy

Algorithm 1 On-policy Monte Carlo Control

Input: ϵ random action probability, γ discount factor
$\pi \leftarrow$ e-greedy policy w.r.t $Q(s,a)$
Initialize $Q(s,a)$ arbitrarily, with $Q(terminal,\cdot) = 0$
$G(s,a) \leftarrow [\,]$
for episode $\in 1..N$ **do**
    Generate episode following $\pi : S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    for $t \in T-1..0$ **do**
        $G \leftarrow R_{t+1} + \gamma G$
        Append G to $G(S_t, A_t)$
        $Q(s,a) \leftarrow average(G(S_t, A_t))$
    end for
end for
Output: Near optimal policy π and action values $Q(s,a)$

On-policy Monte Carlo Control

Import the necessary software libraries:

```
import numpy as np
import matplotlib.pyplot as plt

from envs import Maze
from utils import plot_policy, plot_values, test_agent
```

Initialize the environment

```
env = Maze()
```

```
frame = env.render(mode='rgb_array')
plt.figure(figsize=(4,4))
plt.axis('off')
plt.imshow(frame)
```

```
print(f"Observation space shape: {env.observation_space.nvec}")
print(f"Number of actions: {env.action_space.n}")
```

```
Observation space shape: [5 5]
Number of actions: 4
```

On-policy Monte Carlo Control

Define the policy $\pi(s)$

Create the policy $\pi(s)$

```
def policy(state, epsilon=0.):
    if np.random.random() < epsilon:
        return np.random.randint(4)
    else:
        av = action_values[state]
        return np.random.choice(np.flatnonzero(av == av.max()))
```

Test the policy with state (0, 0)

```
action = policy((0,0))
print(f"Action taken in state (0,0): {action}")
```

```
Action taken in state (0,0): 1
```

Plot the policy

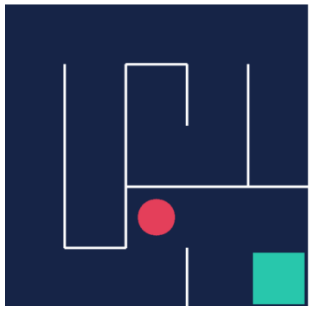
```
plot_policy(action_values, frame)
```

On-policy Monte Carlo Control

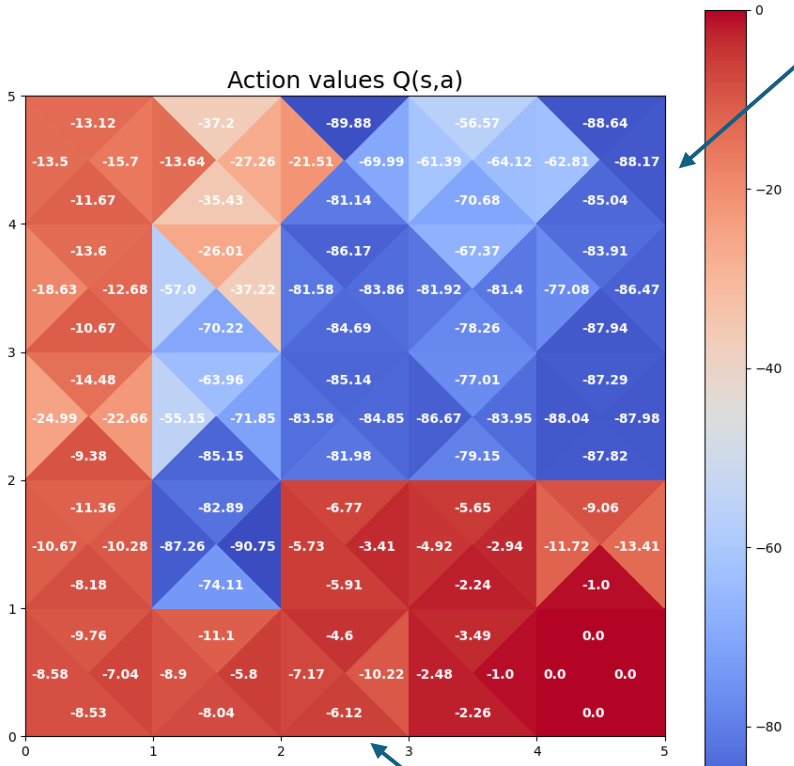
```
def on_policy_mc_control(policy, action_values, episodes, gamma=0.99, epsilon=0.2):

    sa_returns = {}

    for episode in range(1, episodes+1):
        state = env.reset()
        done = False
        transitions = []

        while not done:
            action = policy(state, epsilon)
            next_state, reward, done, _ = env.step(action)
            transitions.append([state, action, reward])
            state = next_state

        G = 0
        for state_t, action_t, reward_t in reversed(transitions):
            G = reward_t + gamma * G

            if not (state_t, action_t) in sa_returns:
                sa_returns[(state_t, action_t)] = []
            sa_returns[(state_t, action_t)].append(G)
            action_values[state_t][action_t] = np.mean(sa_returns[(state_t, action_t)])
```

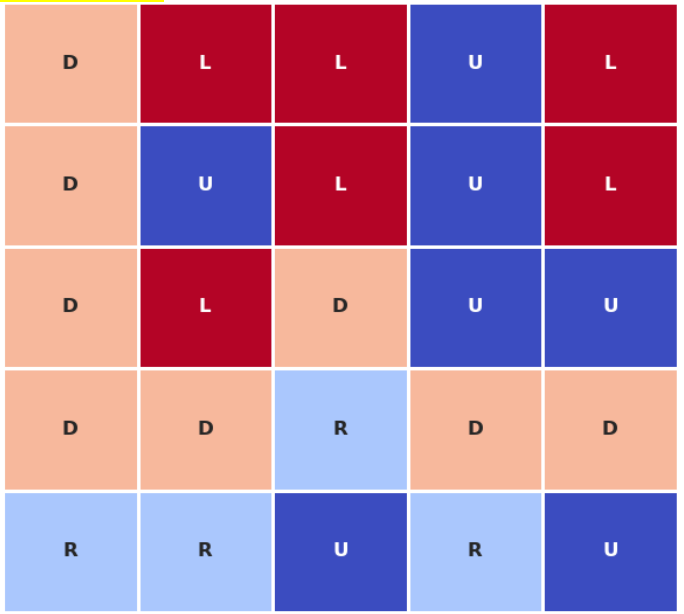
```
on_policy_mc_control(policy, action_values, episodes=10000)
```

On-policy Monte Carlo Control



States off from the optimal path were not updated much





Q(s, a)

Optimal actions are the ones that lead us straight to the goal (on the optimal path)

$\pi(a \mid s)$

Code Ex.

- Open ‘on_policy_control.ipynb’ with jupyter notebook

Off-policy Monte Carlo

Problem with On-policy strategy

- Note On-policy strategy has no REUSE.
- It collects episode, use it to update, and throw it away.

- Data inefficient. Could we reuse old data ?

Besides, in practice there's need for parallel learning, where a group of agent has a common behavior policy but each agent has a different target policy to update their own experience differently (or to pursue multiple optimal solutions)

Off-policy strategy

- Exploration Policy:

$$b(a \mid s)$$

Generates the episode we are going to update Q(s, a) with:

$$S_0, A_0, R_1, S_1, A_1, \ldots, R_T$$

- Target Policy:

$$\pi(a \mid s)$$

Policy to be optimized though Q(s, a) values:

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

Off-policy strategy

- Exploration Policy, b, should be able to explore all the actions that π can take:

$$\text{If } \pi(a \mid s) > 0, \text{ then } b(a \mid s) > 0$$

- The average return will NOT approximate the value under π but under b :

$$E_b[\, G_t \mid S_t = s, A_t = a] = q_b(s, a)$$

Importance sampling

- Statistical technique for estimating the expected values of a distribution by working with samples from another distribution

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

- Recalling Monte Carlo, we can estimate this with samples from q

$$E_q\left[\frac{p(x)}{q(x)}f(x)\right] \approx \frac{1}{N}\sum_{i=1}^{n}\frac{p(x)}{q(x)}f(x_i) \qquad x_i \sim q(x)$$

Importance sampling

- Multiply Wt to correct the return.

$$W_t = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

the prob. of generating the trajectory that produced that return following the target policy, divided by the prob. of generating that return, following the exploratory

- By correcting the returns using Importance Sampling(IS), we will approximate the value under π

$$E[W_t G_t | S_t = s] = v_\pi(s)$$

Update rule

- For each Q(s, a), we'll keep a list of observed returns

$$[G_1, G_2, G_3, \ldots, G_n]$$

- Each time we need to update Q(s, a), we'll recompute the average:
  - $Q(s, a) \leftarrow \frac{1}{N} \sum_{k=1}^{N} G_k$

- Instead, constant α update for smooth learning process:
  - $Q(s, a) \leftarrow Q(s, a) + \frac{W_t}{C(s,a)}[G - Q(s, a)]$,where $C(s, a) = \sum_{k=1}^{N} W_k$

Algorithm 2 Off-policy Monte Carlo Control

Input: γ discount factor
$\pi \leftarrow$ greedy policy w.r.t $Q(s,a)$
$b \leftarrow$ arbitrary policy with coverage of π
$C(s,a) \leftarrow 0$
Initialize $Q(s,a)$ arbitrarily
for episode $\in 1..N$ **do**
    Generate episode following $b : S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    for $t \in T-1..0$ **do**
        $G \leftarrow R_{t+1} + \gamma G$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
        if **then** $A_t \neq \pi(S_t)$
            Break the loop, move to next episode.
        end if
        $W \leftarrow W \frac{1}{b(A_t|S_t)}$
    end for
end for
Output: Optimal π and action values $Q(s,a)$

If the action picked by the exploratory policy is not the same as the action that the target policy would have picked after being updated, (different Ak btw. π and b), we can't calculate $\prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$ anymore

Update W,
by multiplying $\frac{\pi(A_k|S_k)}{b(A_k|S_k)}$
π is greedy policy, where prob. of argmax action is 1

Off-policy Monte Carlo Control

Import the necessary software libraries:

```
import numpy as np
import matplotlib.pyplot as plt

from envs import Maze
from utils import plot_policy, plot_values, test_agent
```

Initialize the environment

```
env = Maze()
```

```
frame = env.render(mode='rgb_array')
plt.figure(figsize=(4,4))
plt.axis('off')
plt.imshow(frame)
```

```
print(f"Observation space shape: {env.observation_space.nvec}")
print(f"Number of actions: {env.action_space.n}")
```

```
Observation space shape: [5 5]
Number of actions: 4
```

Off-policy Monte Carlo Control

Define the target policy $\pi(s)$

Create the policy $\pi(s)$

```
def target_policy(state):
    av = action_values[state]
    return np.random.choice(np.flatnonzero(av == av.max()))
```

Test the policy with state (0, 0)

```
action = target_policy((0,0))
print(f"Action taken in state (0,0): {action}")
```

Create the policy $b(s)$

```
def exploratory_policy(state, epsilon=0.):
    if np.random.random() < epsilon:
        return np.random.randint(4)
    else:
        av = action_values[state]
        return np.random.choice(np.flatnonzero(av == av.max()))
```

Test the policy with state (0, 0)

```
action = exploratory_policy((0,0))
print(f"Action taken in state (0,0): {action}")
```

```
Action taken in state (0,0): 1
```

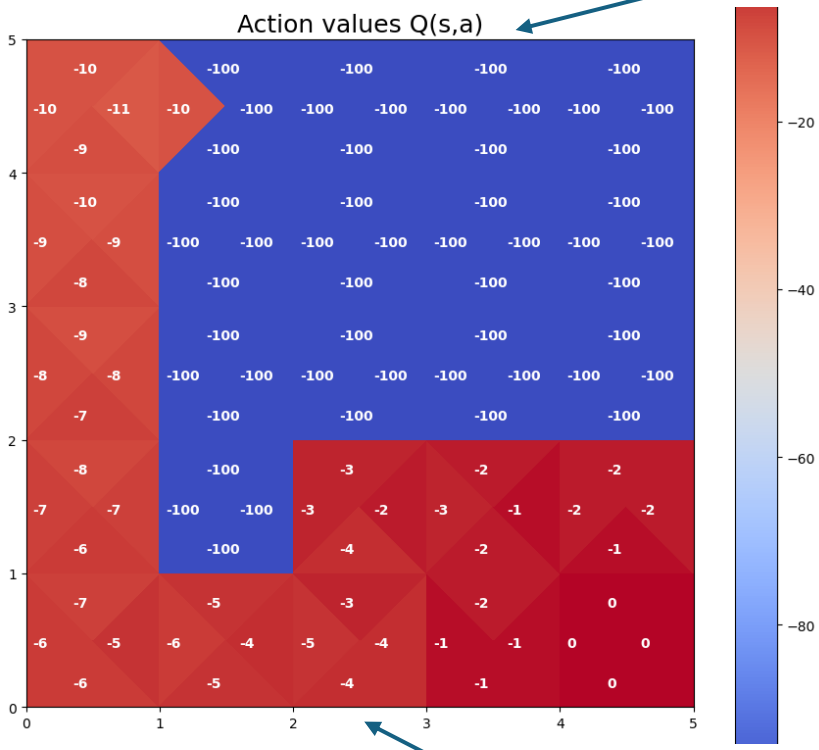
Off-policy Monte Carlo Control

```
def off_policy_mc_control(action_values, target_policy, exploratory_policy, episodes, gamma=0.99, epsilon=0.2):

    for episode in range(1, episodes + 1):
        G = 0
        W = 1
        csa = np.zeros((5, 5, 4))
        state = env.reset()
        done = False
        transitions = []

        while not done:
#             env.render()
            action = exploratory_policy(state, epsilon)
            next_state, reward, done, _ = env.step(action)
            transitions.append([state, action, reward])
            state = next_state

        for state_t, action_t, reward_t in reversed(transitions):
            G = reward_t + gamma * G
            csa[state_t][action_t] += W
            qsa = action_values[state_t][action_t]
            action_values[state_t][action_t] += (W / csa[state_t][action_t]) * (G - qsa)

            if action_t != target_policy(state_t):
                break

            W = W * 1. / (1 - epsilon + epsilon/4)
```

```
off_policy_mc_control(action_values, target_policy, exploratory_policy, episodes=1000, epsilon=0.3)
```

Off-policy Monte Carlo Control

The algorithm hasn't spent much time and effort in refining the estimates of those states that do not lead to the goal(Ignored states on suboptimal paths)



D	L	U	U	U
D	U	U	U	U
D	U	U	U	U
D	U	R	R	D
R	R	U	R	U

Optimal actions are the ones that lead us straight to the goal (highest value than on-policy

Off-policy Monte Carlo Control

- Recall Dynamic Programing sweeps through the whole states space, improving each and every one of them.
  - Although that give us the optimal policy and q-table, it's extremely inefficient because we are wasting a lot of time in states that won't help us achieve our goals

- One of advantages of using methods to learn based on experience is that we can focus or efforts on states and actions that lead us to solving the task in the optimal way
  - For the states that do lead us to the goal, the optimal actions have the highest values
  - Our algorithm didn't spend much effort refining the estimates of those state that do not lead to the goal

Code Ex.

- Open ‘off_policy_control.ipynb’ with jupyter notebook