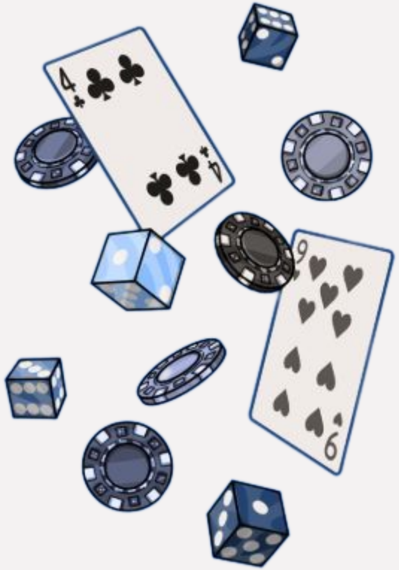
Monte Carlo
Reinforcement Learning



Gachon University
Department of AI/SW
202135520 김승수 (Kim SeungSu)

Table of Contents

Monte Carlo Method

"Computational algorithms that rely on repeated random sampling to obtain numerical results"

- Wikipedia

Monte Carlo Method

How to obtain winrate of Solitaire?

1. Calculate odds

possible outcomes = 52! ≈ 8*10^67
=> not feasible

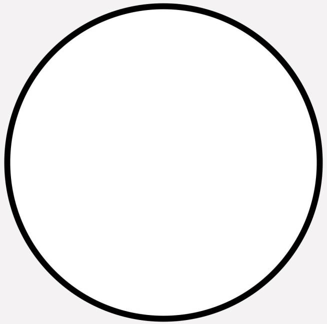
2. Estimate

play N games and see how many times are wins
=>much easier

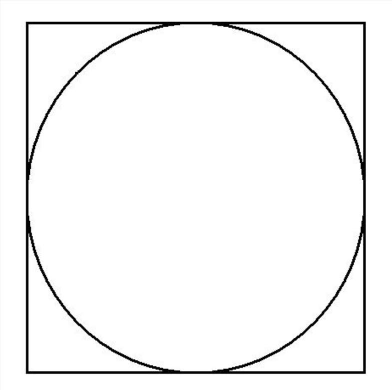
Monte Carlo Method example

How to get value of $\boldsymbol{\pi}$?

1. Draw a circle



2. Draw square around circle with length equal to diameter

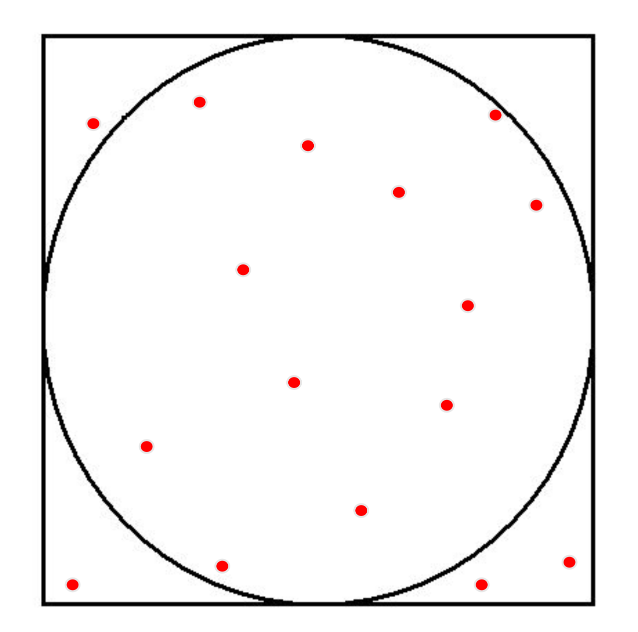


Monte Carlo Method example

3. Put N points inside the square
- N = 16

4. Count points inside circle (k)
- k = 12

5. Area of circle / area of square
$= \pi * r * r \ / \ 2r * 2r = \pi/4$
- $12/16 = 0.75 \approx 0.7853... = \pi/4$



Background knowledge

1. Return

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

2. Policy
- represented as $\boldsymbol{\pi}$
- maps states to probabilities of selecting each possible action
- what we want to learn

$$\max_{\pi} \mathbb{E}_{\pi}[G_t]$$

Background knowledge

3. Value function
- state-value function: expected return when starting in state s and following $\boldsymbol{\pi}$

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right], \text{ for all } s \in \mathcal{S}$$

- action-value function: expected return when starting in state s, performing action p and following $\boldsymbol{\pi}$

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right]$$

Background knowledge

3. Optimal policy
- represented as ***π****

4. Optimal state-value function
- represented as v*
- used to get ***π****

$$v_*(s) \doteq \max_\pi v_\pi(s), \text{ for all } s \in \mathcal{S}$$

5. Optimal action-value function
- represented as q*
- used to get ***π****

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a), \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}(s)$$

Monte Carlo Reinforcement Learning

Characteristics:

1. Learns from episode of experience
2. Model-free
3. Uses idea that value function = empirical mean return
4. When choosing action, randomly choose between exploration and exploitation to prevent local minima

Monte Carlo Reinforcement Learning

Mean of x_1 ~ x_k:

$$\mu_k = \frac{1}{k}\sum_{j=1}^{k} x_j \quad = \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j\right)$$
$$= \frac{1}{k}(x_k + (k-1)\mu_{k-1})$$
$$= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$$

Say x= g

$\mu_k = E(x) = E(g_t \mid S_t = s) = v(S_t)$

Thus, where $N(S_t)$ represents the amount of time a state was visited:

$v(S_t) := v(S_t) + 1/N(S_t)*(g_t - v(S_t))$

This can be used to update the policy

ε- greedy exploration pseudo code

```
For episode in num_episode:
    While not done:
        If rand(0, 1) >= epsilon:
            action = argmax(Q[state, :]) # exploit
        else:
            action = rand(actions) # explore
        new_state,reward,done = action.perform()
        state = new_state
        results_list += (state, action)
        results_sum += reward

    For (state, action) in  results_list:
        visit[state, action] += 1
        alpha = 1/visit[state,action]
        Q[state,action] += alpha*(result_sum - Q[state, action])
```

ε- greedy exploration walkthrough (0)

$\varepsilon = 0.2$
State = x
Actions = u(p), d(own), l(eft), r(ight)

Starting point	Reward: 0 Done: X	Reward: 0 Done: X
Reward: 0 Done: X	Reward:10 Done: O	Reward: 0 Done: X
Reward: 0 Done: X	Reward: 0 Done: X	Reward: 100 Done: O

ε- greedy exploration walkthrough (1)

$\varepsilon = 0.2$
State = (0, 0)
Actions = u(p), d(own), l(eft), r(ight)

rand() = 0.5 >= ε
argmax(Q[(0,0), :]) = r

Starting point S	Reward: 0 Done: X	Reward: 0 Done: X
Reward: 0 Done: X	Reward:10 Done: O	Reward: 0 Done: X
Reward: 0 Done: X	Reward: 0 Done: X	Reward: 100 Done: O

ε- greedy exploration walkthrough (2)

ε = 0.2
State = (1, 0)
Actions = u(p), d(own), l(eft), r(ight)

rand() = 0.1 < ε
rand(actions) = d

Starting point	Reward: 0 Done: X S	Reward: 0 Done: X
Reward: 0 Done: X	Reward:10 Done: O	Reward: 0 Done: X
Reward: 0 Done: X	Reward: 0 Done: X	Reward: 100 Done: O

ε- greedy exploration walkthrough (3)

$\varepsilon = 0.2$
State = (1, 0)
Actions = u(p), d(own), l(eft), r(ight)

Done = true
Reward = 10

Starting point	Reward: 0 Done: X	Reward: 0 Done: X
Reward: 0 Done: X	Reward:10 Done: O s	Reward: 0 Done: X
Reward: 0 Done: X	Reward: 0 Done: X	Reward: 100 Done: O

ε- greedy exploration walkthrough (4)

Visit[(0, 0), r] = 0 +1 = 1
Q[(0, 0), r] = 0 + 1/1*(10-0) = 10

visit[(1, 0), d] = 0 + 1 = 1
Q[(1, 0), d] = 0 + 1/1*(10-0) = 10

Starting point	Reward: 0 Done: X	Reward: 0 Done: X
Reward: 0 Done: X	Reward:10 Done: O	Reward: 0 Done: X
Reward: 0 Done: X	Reward: 0 Done: X	Reward: 100 Done: O

ε- greedy exploration walkthrough (5)

ε = 0.2
State = (0, 0)
Actions = u(p), d(own), l(eft), r(ight)

rand() = 0.7 >= ε
argmax(Q[(0,0),:]) = r

Starting point s	Reward: 0 Done: X	Reward: 0 Done: X
Reward: 0 Done: X	Reward:10 Done: O	Reward: 0 Done: X
Reward: 0 Done: X	Reward: 0 Done: X	Reward: 100 Done: O

ε- greedy exploration walkthrough (6)

$\varepsilon = 0.2$
State = (1, 0)
Actions = u(p), d(own), l(eft), r(ight)

rand() = 0.1 < ε
rand(actions) = r

Starting point	Reward: 0 Done: X S	Reward: 0 Done: X
Reward: 0 Done: X	Reward:10 Done: O	Reward: 0 Done: X
Reward: 0 Done: X	Reward: 0 Done: X	Reward: 100 Done: O

ε- greedy exploration walkthrough (7)

ε = 0.2
State = (2, 0)
Actions = u(p), d(own), l(eft), r(ight)

rand() = 0.5 >= ε
argmax(Q[(2,0),:]) = d

Starting point	Reward: 0 Done: X	Reward: 0 Done: X s
Reward: 0 Done: X	Reward:10 Done: O	Reward: 0 Done: X
Reward: 0 Done: X	Reward: 0 Done: X	Reward: 100 Done: O

ε- greedy exploration walkthrough (8)

ε = 0.2
State = (2, 1)
Actions = u(p), d(own), l(eft), r(ight)

rand() = 0.9 >= ε
argmax(Q[(2,1),:]) = d

Starting point	Reward: 0 Done: X	Reward: 0 Done: X
Reward: 0 Done: X	Reward:10 Done: O	Reward: 0 Done: X S
Reward: 0 Done: X	Reward: 0 Done: X	Reward: 100 Done: O

ε- greedy exploration walkthrough (9)

ε = 0.2
State = (2, 2)
Actions = u(p), d(own), l(eft), r(ight)

Done = 100
Reward = 100

Starting point	Reward: 0 Done: X	Reward: 0 Done: X
Reward: 0 Done: X	Reward:10 Done: O	Reward: 0 Done: X
Reward: 0 Done: X	Reward: 0 Done: X	Reward: 100 Done: O s

ε- greedy exploration walkthrough (10)

visit[(0, 0), r] = 1+1 = 2
Q[(0, 0), r] = 10 + 1/2*(100-10) = 55

visit[(1, 0), r] = 0 + 1 = 1
Q[(1, 0), r] = 0 + 1/1*(100-0) = 100

visit[(2, 0), d] = 0 + 1 = 1
Q[(2, 0), d] = 0 + 1/1*(100-0) = 100

visit[(2, 1), d] = 0 + 1 = 1
Q[(2, 1), d] = 0 + 1/1*(100-0) = 100

Starting point	Reward: 0 Done: X	Reward: 0 Done: X
Reward: 0 Done: X	Reward:10 Done: O	Reward: 0 Done: X
Reward: 0 Done: X	Reward: 0 Done: X	Reward: 100 Done: O

ε- greedy exploration python code (1)

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

def random_action(actions, state, row_size, column_size):
    while(True):
        # choose random action
        action = random.choice(actions)

        # make sure action is possible
        if action == "up":
            if state[0] > 0:
                return 0
        elif action == "down":
            if state[0] < row_size -1:
                return 1
        elif action == "left":
            if state[1] > 0:
                return 2
        elif action == "right":
            if state[1] < column_size - 1:
                return 3

def perform_action(action, state, rewards):
    new_state = state.copy()
    if action == 0:
        new_state[0]-=1
    elif action == 1:
        new_state[0]+=1
    elif action == 2:
        new_state[1]-=1
    elif action == 3:
        new_state[1]+=1

    reward = got_reward(new_state, rewards)
    if reward != 0:
        return new_state, reward, True
    else:
        return new_state, 0, False

def got_reward(state, rewards):
    for reward in rewards:
        if (state[0],state[1]) == reward[0]:
            return reward[1]

    return 0
```

```
def visualize_path(results_list, row_size, column_size):
    grid = np.zeros((row_size, column_size))
    for state, action in results_list:
        grid[state[0]][state[1]] = action + 1

    print("="*100)
    print(len(results_list))
    for r in range(row_size):
        for c in range(column_size):
            if grid[r][c] == 0:
                print(". ", end = "")
            elif grid[r][c] == 1:
                print("↑ ", end = "")
            elif grid[r][c] == 2:
                print("↓ ", end = "")
            elif grid[r][c] == 3:
                print("← ", end = "")
            elif grid[r][c] == 4:
                print("→ ", end = "")
        print("")
    print("="*100)
actions = ["up", "down", "right", "left"]
# parameters ============
row_size = 5
column_size = 5
epsilon = 0.3
num_episodes = 20000
rewards = [((1,1), 3), ((0,3), 70), ((4,4),100000), ((2, 3), 5000)]
# ======================
avg_reward_per_action = [0] *5
episode_count = [0]*5

Q = np.zeros((row_size,column_size, len(actions)))
visit_count = np.zeros((row_size, column_size, len(actions)))
```

```
for episode in range(num_episodes):
    # reset variables
    state = [0, 0]
    result = 0
    results_list = []
    results_sum = 0
    reward = 0
    done = False
    count = 0

    # complete episode
    while not done:
        if np.random.rand() > epsilon:
            action = np.argmax(Q[state[0], state[1], :])
            # if max is 0, choose randomly
            if Q[state[0], state[1], action] == 0:
                action = random_action(actions, state, row_size, column_size)
        else:
            action = random_action(actions, state, row_size, column_size)
        new_state, reward, done = perform_action(action, state, rewards)
        results_list.append((state, action))
        state = new_state.copy()
        results_sum += reward
        count+=1

    # save values
    index = math.floor(math.log10(episode+1))
    episode_count[index]+=1
    avg_reward_per_action[index] += 1/episode_count[index]*(reward/count -
avg_reward_per_action[index])
    #print(f"{episode}: {reward/count}")

    # update Q matrix
    for (state, action) in results_list:
        visit_count[state[0], state[1], action] += 1
        alpha = 1/visit_count[state[0], state[1],action]
        Q[state[0], state[1], action] += alpha*(results_sum - Q[state[0], state[1],
action])

    # print path
    visualize_path(results_list, row_size, column_size)

# print results
for i in range(len(avg_reward_per_action)):
    print(f"range: ~10^{i+1} - average reward per action: %.3f data size: %d"
%(avg_reward_per_action[i], episode_count[i]))

# visualize results
plt.bar(["1~9", "10~99", "100~9999", "10000~99999", "100000~999999"],
avg_reward_per_action)
plt.show()
```

ε- greedy exploration python code (2)

Current version (my Google drive):
https://drive.google.com/file/d/1GcvEHfC8JNfFg516VT7YL4QQbFkZBfFw/view?usp=sharing

Newest version (my Github):
https://github.com/DeceptiveRat/Monte-Carlo-Reinforcement-Learning

ε- greedy exploration python results (1)

At first, the agent goes to nearby rewards and takes long paths

```
5
→ → ↓ . .
. . → ↑ .
. . . . .
. . . . .
. . . . .
==========
==========
4
↓ . . . .
↓ . . . .
→ ↑ . . .
. . . . .
. . . . .
==========
==========
10
→ → ↓ . .
. . ↓ . .
. ↑ ↓ . .
. ↑ ← . .
. . ↑ . .
```

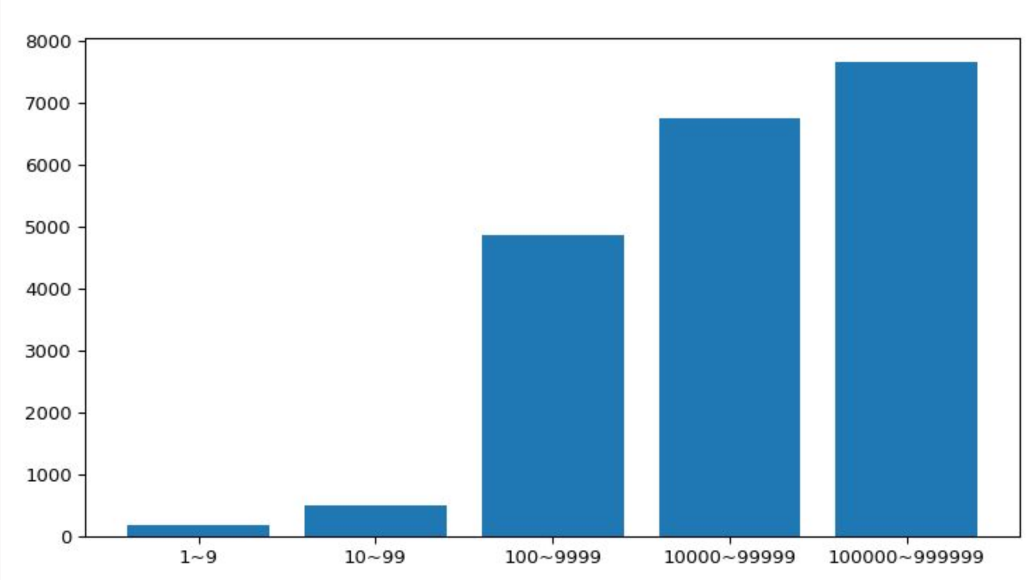
Later, the agent takes short paths to rewards far away

```
16
↓ ← . . .
↓ . . . .
↓ . . . .
↓ . . . .
→ → → → .
==========
==========
8
↓ . . . .
↓ . . . .
↓ . . . .
↓ . . . .
→ → → → .
==========
==========
10
↓ ← . . .
↓ . . . .
↓ . . . .
↓ . . . .
→ → → → .
==========
==========
8
↓ . . . .
↓ . . . .
↓ . . . .
↓ . . . .
→ → → → .
```

ε- greedy exploration python results (2)

```
range: ~10^1 - average reward per action: 160.338 data size: 9
range: ~10^2 - average reward per action: 497.321 data size: 90
range: ~10^3 - average reward per action: 4861.334 data size: 900
range: ~10^4 - average reward per action: 6736.648 data size: 9000
range: ~10^5 - average reward per action: 7657.695 data size: 10001
```

Average reward per action can be seen going up as episodes pass



Pros and Cons

Pros:
- doesn’t require a model
- conceptually simple
- easy to implement

Cons:
- inefficient for long episodes
- requires episodes to be finished to learn from it

References

1. Durham University Lectures: https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://cwkx.github.io/data/teaching/dl-and-rl/rl-lecture5.pdf&ved=2ahUKEwi99cWsoaiQAxWCsVYBHQtuNhwQFnoECB4QAQ&usg=AOvVaw1OS2PUtCnzYGNTCOr3Ecks

2. Wikipedia: https://en.wikipedia.org/wiki/Monte_Carlo_method

3. Medium: https://medium.com/@hsinhungw/intro-to-reinforcement-learning-monte-carlo-to-policy-gradient-1c7ede4eed6e

THANK YOU