HW1 report

Question 1

這一題的目標簡單來說就是要完成簡報當中提及的 Iterative policy evaluation。

Iterative policy evaluation

Input π, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$

所以照著這一個演算法流程可以製作出底下的函數。
演算法當中的 Δ 目的是要去計算出新的 V 跟舊的 V 之間有多大的差異。這裡不直接儲存 Δ，而是單純計算新舊之間的差異。當之間的差異已經比設定的 ϵ 小就會結束。

我們會把新的 V 存放在 `V_new` 當中，透過一個迴圈去看過當前這個 state 底下的所有 action，找到 `V_new` 的所有數值。

```python
def policy_evaluation(self, epsilon=1e-6):
    self.V = np.zeros(self.grid_size)
    while True:
        V_new = np.zeros_like(self.V)
        for s in range(self.grid_size):
            if s == 0 or s == self.grid_size - 1:
                continue
            v_s = 0
            for action, _ in self.actions.items():
                s_prime = self.transition(s, action)
                v_s += 0.25 * (self.get_reward(s) + self.discount_factor * self.V[s_prime])
            V_new[s] = v_s
        if np.max(np.abs(V_new - self.V)) < epsilon:
            break
        self.V = V_new
        self.history.append(self.V.copy())
    return self.V
```

如此一來就完成了。

```python
if __name__ == "__main__":
    gammas = [1, 0.9, 0.1]
    for gamma in gammas:
        grid_world = GridWorld(discount_factor=gamma)
        state_values = grid_world.policy_evaluation()
        print(f'========== Gamma = {gamma} ==========')
        print(state_values)
        np.savetxt(f'110062126_hw1_1_data_gamma_{gamma}', state_values[1:-1], fmt='%.2f', delimiter=' '
```

不同的 γ 會使得最後得到的 Value 每個數值的絕對值。γ 越大，數值的絕對值也會越大。並且每個 state 的 value 差異也會被拉大。

```
========== Gamma = 0.9 ==========
[ 0.         -5.27780985 -7.12839461 -7.65050301 -5.27780985 -6.60628621
 -7.18060555 -7.12839461 -7.12839461 -7.18060555 -6.60628621 -5.27780985
 -7.65050301 -7.12839461 -5.27780985  0.        ]
========== Gamma = 0.1 ==========
[ 0.         -1.0825649  -1.11037455 -1.11107145 -1.0825649  -1.10967766
 -1.11100184 -1.11037455 -1.11037455 -1.11100184 -1.10967766 -1.0825649
 -1.11107145 -1.11037455 -1.0825649   0.        ]
```

Question 2

這一題的目標是要使用 SARSA 以及 Q-learning 來解 Taxi 問題。

SARSA

目標是就是實作講義當中的演算法。

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

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\text{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

實作如下。

1. 初始化 $Q(s,a)$
   方便起見全部都設為 0 就不需要處理 terminal state 的部份。
2. 重複底下的操作，這裡 episode 數量設定為 10000
   - 初始化 S
   - 透過 ϵ-greedy 去選 action A
   - 重複底下操作直到 Terminate
     - 執行 A，得到 R, S'
     - 透過 ϵ-greedy 去選 action A'

- 更新 $Q(S, A)$

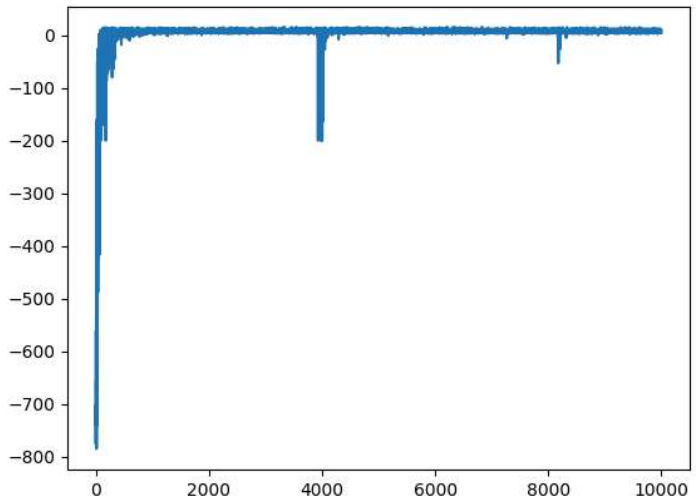
```python
rewards = []
Q = np.zeros((num_state, num_action))

# Training
for _ in trange(10000):
    # Initialize S
    S, info = env.reset()
    # Choose A from S using epsilon greedy
    A = epsilon_greedy(Q, S)
    terminated = truncated = False
    # Total return for recording
    total_reward = 0
    while not (terminated or truncated):
        # Take action A, observe R, S'
        S_prime, reward, terminated, truncated, info = env.step(A)
        # Choose A' from S' using epsilon greedy
        A_prime = epsilon_greedy(Q, S_prime)
        # update Q-table
        Q[S, A] = Q[S, A] + alpha * (reward + gamma * Q[S_prime, A_prime] - Q[S, A])
        # update S and A
        S, A = S_prime, A_prime
        # update total_reward
        total_reward += reward
    rewards.append(total_reward)
env.close()
```

詳細的 Hyperparameter 設定如下。

```
epsilon = 0.0001
alpha = 0.8
gamma = 0.9
```

當時沒有特別挑選這些 hyperparameter，很快就收斂了。



> 橫軸為 episode number，縱軸為 total reward。
> 可以觀察到基本上從 4000 之後都很平穩了。

Q-learning

目標就是實作講義當中的演算法。

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

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

實作如下。

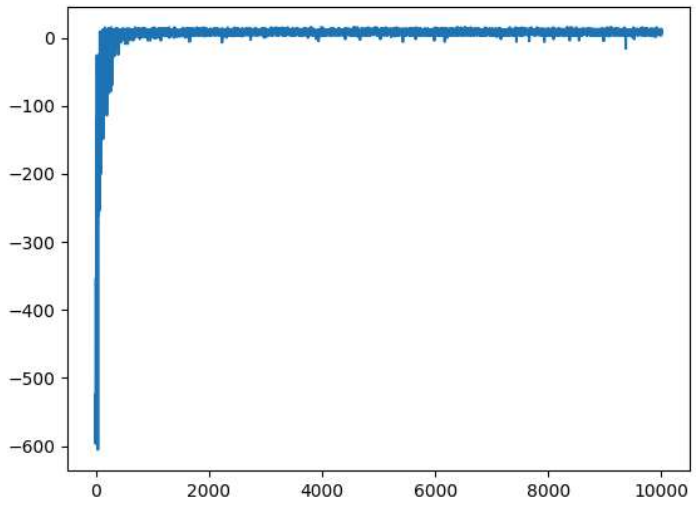
1. 初始化 $Q(S, A)$
   同樣方便起見一開始大家都設為 0 就不需要額外處理 terminal state。
2. 重複底下的操作，這裡設定的 episode 數量為 10000
   - 初始化 S
   - 重複 terminate
     - 透過 ϵ-greedy 選一個 action A
     - 取得 R, S'
     - 更新 $Q(S, A)$

```
rewards = []
Q = np.zeros((num_state, num_action))

for _ in trange(10000):
    # Initialize S
    S, info = env.reset()
    terminated = truncated = False
    # Total return for recording
    total_reward = 0
    while not (terminated or truncated):
        # Choose A from S using epsilon greedy
        A = epsilon_greedy(Q, S)
        # Take action A, observe R, S'
        S_prime, reward, terminated, truncated, info = env.step(A)
        # update Q-table
        Q[S, A] = Q[S, A] + alpha * (reward + gamma * np.max(Q[S_prime, :]) - Q[S, A])
        # update S
        S = S_prime
        # update total_reward
        total_reward += reward
    rewards.append(total_reward)
env.close()
```

詳細的 Hyperparameter 如下。

```
epsilon = 0.001
alpha = 0.7
gamma = 0.6
```



> 橫軸為 episode number，縱軸為 total reward。
> 可以觀察到基本上從 2000 之後都很平穩了。

Question 3

這一題我選擇用 Q-learning 處理。因此最核心的部分跟 Question 2 的 Q-learning 相同。

```
def training(env, agent, iters=1000000):
    for iter in trange(iters):
        init_player = [O_num, X_num][random.randint(0, 1)]
        player = init_player
        # Initialize S
        S = env.reset(player)
        terminated = False

        while not (terminated):
            # Choose A from S using epsilon greedy
            A = agent.epsilon_greedy(player, S)
            # Take action A, observe R, S'
            S_prime, reward, terminated = env.step(A, agent)
            # update Q-table
            agent.update_policy(player, S, A, S_prime, reward)
            # update S
            S = S_prime

        if iter % 1000 == 0 and iter != 0:
            agent.save_policy()
```

實際上訓練的 agent 只有一個。訓練過程中的對手其實也是 agent 本身。為了實作上方便，agent 只會學習下 o 的狀況，如果接下來要下的是 x ，那就把整個盤面的 o , x 互換，問這個狀況 o 會下在哪裡就跟原問題相同。

自從第二題看到 OpenAI Gym 處理的方法後覺得有 `Env` 去描述環境，有 `Agent` 來跟環境互動，這樣的設定會使 code 更簡單明瞭，因此在這裡我也嘗試做到類似的事情。

Environment

對於 `Env` 來說，最重要的是 `reset()` 以及 `step()` 這兩個函數。

- `reset()`
  要去把環境 reset 到一個合法的 state，準備開始新的一局。
- `step()`
  在現在這個 state 底下去執行 action `A`，將觀察到的下一個 state、reward、是否 terminate 回傳。

reset()

`reset()` 的部份還蠻簡單的，麻煩的地方只在於要產生出一個合法的盤面，因此有額外寫了一些函數協助判斷，但邏輯上就是找到合法盤面後回傳。

```
def reset(self, current_player):
    self.current_player = current_player
    while True:
        self.current_state_num = random.randint(0, num_states-1)
        self.current_state = copy.deepcopy(to_state(self.current_state_num))
        if self.is_valid(self.current_state, self.current_player) and not self.is_terminal_state(self.current_state):
            break
    return self.current_state_num
```

> 所謂的合法指的是下一步是 `current_player` 要下，無論在他下之前、之後都不會出現底下兩個狀況。
>
> 1. 有多個人獲勝
> 2. 有任一方下棋的次數相差超過 1

step()

`step()` 當中最重要的就是 **transition**、**決定 reward**、**判斷 teminate** 這三件事。

trasition 就單純把指定的位置填上玩家的 `O` 或 `X`。

Reward 的決定上比較麻煩一些。最基本的 reward 可以有底下幾種

- 選擇的格子已經被填過了
  這種狀況要極力避免，因此我設定 `-100` 的 reward，並且直接 terminate。
- 對手贏
  這種狀況也要避免，因此設定 `-20` 的 reward，並且直接 terminate。
- 我方贏
  這種狀況是最好的，因此設定 `20` 的 reward，並且會 terminate。
- 平手
  這種狀況不好也不壞，因此設定 `0` 的 reward，並且會 terminate。

不過實際上 **對手贏** 這個狀況要被考慮進來，也就意味著我們在當前的 player 執行 action `A` 之後對手也需要執行一個 action。因此這裡的 `step` 更精確的說法會是執行 action `A` ，且對手接著執行 action `A'` 。

實作細節如下。

選擇的格子已經被填過

```
def step(self, A, agent):
    action = (A//3, A%3)
    # Not a valid action
    bad_actions = [(x, y) for x in range(3) for y in range(3) if self.current_state[x][y] != empty_num]
    if action in bad_actions:
        return self.current_state_num, -100, True
```

我方獲勝

```
self.current_state[action[0]][action[1]] = self.current_player
self.current_state_num = to_state_index(self.current_state)

if self.judge(self.current_state, self.current_player):
    return self.current_state_num, 20, True
```

平手

```
elif self.is_terminal_state(self.current_state):
    return self.current_state_num, 0, True
```

對手再走一步

```
new_action = agent.choose_action(np.array([self.current_player*-1] + np.array(self.current_state).reshape(-1).tolist()))
self.current_state[new_action//3][new_action%3] = self.current_player*-1
self.current_state_num = to_state_index(self.current_state)

next_state, reward, terminated = self.current_state_num, 0, self.is_terminal_state(self.current_state)
```

處理下一步的輸贏

```
if self.judge(self.current_state, self.current_player):
    reward = 20
elif self.judge(self.current_state, self.current_player*-1):
    reward = -20
elif terminated:
    reward = 0
```

Agent

`Agent` 的部份最重要的就是 `choose_action()`, `update_policy()` 這兩個函數。分別是要去依據 policy 選擇最佳 action 以及更新 policy。

choose_action()

由於我們只有訓練下一手是 `o` 的狀況，因此對於問 `x` 的情形需要事先把盤面翻轉過來。接下來就可以從 $Q(S,\cdot)$ 當中找到最大的當成答案。

```
def choose_action(self, query):
    current_player = query[0]
    state = np.array(query[1:]).reshape((3, 3)).tolist()
    state_idx = to_state_index(state)

    if current_player == X_num:
        state = (np.array(state)*(-1)).tolist()
        state_idx = to_state_index(state)

    return np.argmax(self.Q[state_idx])
```

update_policy()

也就是把 Q-learning 更新 policy 的部份寫在這邊。同樣也需要先注意如果要問的是 x 的話需要先把 state 翻轉。

```
def update_policy(self, current_player, S, A, S_prime, reward):
    # update Q-table
    if current_player == X_num:
        current_state = (np.array(to_state(S)) * -1).tolist()
        S = to_state_index(current_state)
    else:
        next_state = (np.array(to_state(S_prime)) * -1).tolist()
        S_prime = next_state
    self.Q[S, A] += self.alpha * (reward + self.gamma * np.max(self.Q[S_prime, :]) - self.Q[S, A])
    self.epsilon -= self.epsilon_decay
```

細節設定

詳細的 Hyperparameter 設定如下。

```
epsilon = 0.5
epsilon_decay = 5e-7
alpha = 0.7
gamma = 0.6
episode = 1000000
```

問題與改善

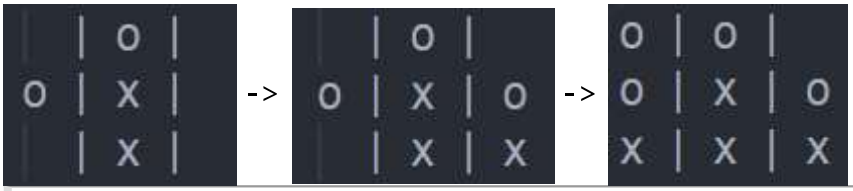
經過訓練之後發現跟自己互打的勝率大約可以到 85%，仔細觀察那些輸掉的盤面可以發現到基本上有兩類。

1. 本來就必輸的局面
   我只有篩掉一步內必輸的狀況，因此 Evaluation 階段還是有一些兩步內必輸的局面出現。

```
O | O | X
X |   |
  | X |
```

> 例如上面的盤面，下一手是 O 。

2. 沒有成功阻擋對方必勝的局面，或把必勝局面玩輸



> 例如上面的盤面，下一手是 o 。
> 原本 o 是必勝的，但卻玩到輸了

第一個並不是問題，針對第二個問題，我認為是因為還沒好好收斂到這一個盤面，因此針對這些輸掉的盤面，我另外讓模型去 "finetune" 他。

在 Training 加上參數 `finetune` 控制。如果要 finetune 的話，除了 sample 一個合法盤面以外，更重要的是找出輸掉的盤面。因此我寫了一個 `can_win()` 來判斷，篩掉這些已經可以贏的部分。

```
def training(env, agent, iters=1000000, finetune=False):
    for iter in trange(iters):
        init_player = [O_num, X_num][random.randint(0, 1)]
        player = init_player
        # Initialize S
        S = env.reset(player)
        terminated = False

        if finetune:
            while can_win(S, player, agent):
                S = env.reset(player)
```

經過這樣的調整後，可以把勝率拉高到 97%~99%。

此外，如果把 ϵ 調高到 0.6，對於前期的訓練可以帶來一些幫助。但因為後續都會經過 finetune，最後得出來的結果基本上都很不錯，實際上就沒有再換成 0.6 了。

Question 4

這一題的目標是要透過 Monte Carlo Tree Search 去實作 3D Tic-Tac-Toe。不過因為之前沒有學過 MCTS，所以底下先說明一下我對於 MCTS 的理解。

MiniMax

理解 MCTS 可以先從 MiniMax 開始思考。
今天在一個 state 當中有許多的 action 可以選擇，要選擇哪個比較好呢？直覺上就可以把所有的結果樹狀展開，找到其中最佳的 action。

不過在實際狀況下以棋類比賽來說，對手會做的決定也不一定會是很糟糕的，如果我們單純去考慮所有狀況下一個 action 帶來的好處，就有可能會把對手根本不會考慮的最差策略也納入考量。

因此 MiniMax 的想法是假設對手很聰明，都會選擇對自己最有利的 action，或反過來說，是對對手最不好的決策。

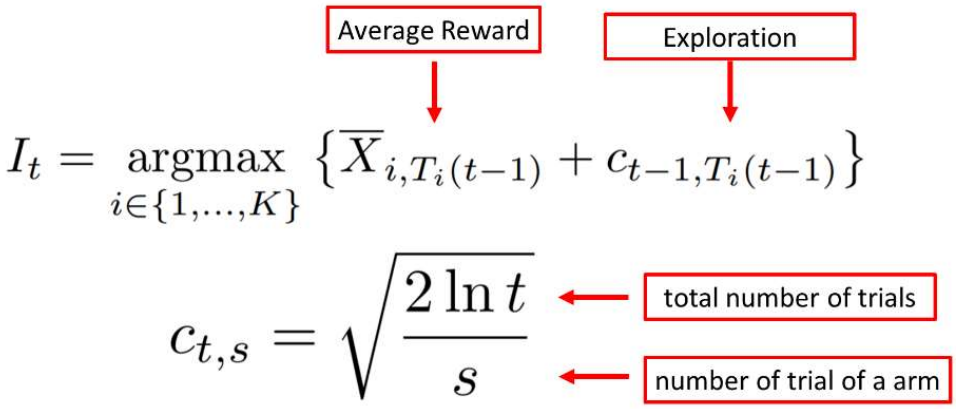
因此在樹上的搜尋過程會像是 (最大化自己的勝率) -> (最小化自己的勝率) -> (最大化自己的勝率) -> ... 這樣不斷重複直到結束。

UCB

然而直接把整個樹狀結構攤開來，在 state, action 很多的情況下是不可能做到的。因此 MCTS 會去估計每個 action 帶來的效益，去評估接下來要去走哪一個 action。

UCB 是其中一種估計方式，會希望去平衡 exploration 以及 exploitation。

- 一個 action 嘗試的次數少，會希望可以多嘗試
- 一個 action 依照過往經驗分數高，也會希望可以多使用



- i: Bandit i
- I_t: 時間 t 選擇使用的 bandit
- K: Bandit 的數量
- $T_i(t-1)$: 在前 $t-1$ 的時間當中 bandit i 被選中的次數
- $\bar{X}_{i,T_i(t-1)}$: 在前 $t-1$ 的時間當中 bandit i 得到的平均 reward

MCTS

MCTS 同樣是在描述一個樹狀結構，不過在 action 的選擇是透過 UCB 等方法去找到估計最佳的。

MCTS 分成了 ***Selection***, ***Expansion***, ***Roll-out***, ***Backpropagation*** 這四個步驟。

- Selection
  從當前的 state 作為 root 往下依照 UCT 決定 action 直到 leaf node。
- Expansion
  從 selection 走到的 leaf node 往下擴張一個 node。
- Roll-out
  從 Expansion 的 node 往後模擬一次直到結束。

- Back Propagation
  將得到的結果回傳到中間經過的每個 nodes。

實作

整體實作上首先處理 TreeNode 的部份，描述一個樹節點會做哪些操作。

UCB 計算

```
UCB(self):
return (self.total_reward / (self.visit_count + 1)) + self.c * (np.sqrt(np.log(self.parent.visit_count + 1)) / (self.visit_count + 1))
```

Select

單一節點根據 UCB 找到最好的 action。

```
def select(self):
    """
    Select the best action according to UCB
    Return (action, TreeNode)
    """
    if self.is_leaf_node():
        return -1
    return max(self.children.items(), key=lambda child: child[1].UCB())
```

Expand

從當前的節點向下 expand。

```
def expand(self, actions):
    """
    Expand actions from current node
    """
    for action in actions:
        if action not in self.children:
            self.children[action] = TreeNode(self)
```

Update

更新節點的 total_reward。

```
def update(self, reward):
    self.total_reward += reward
```

Backpropagate

向上更新，將 reward 傳回。

```
def backpropagate(self, reward):
    self.update(reward)
    if self.parent != None:
        self.parent.update(reward)
```

接下來就可以處理整個 MCTS 的部份。

Play_once

嘗試跑 MCTS 一輪，也就是經過 selection, expansion, roll-out, backpropagate 四個步驟。

```
def play_once(self, state:Board):
    """
    Try to perform MCTS once, return reward
    """
    # Selection
    node = self.root
    while not node.is_leaf_node():
        action, node = node.select()
        state.do_move(action)
    # Expansion
    end, winner = state.is_terminate()
    if not end:
        node.expand(state.availables)
    # Roll-out
    reward = self.roll_out(state)
    # Backpropagate
    node.backpropagate(reward)
```

roll-out

實際 Roll-out，並且回傳 reward。

```
def roll_out(self, state:Board):
    """
    Roll out, return reward
    """
    current_player = state.get_current_player()
    end, winner = state.is_terminate()
    while not end:
        action = state.availables[np.random.randint(0, len(state.availables))]
        state.do_move(action)
        end, winner = state.is_terminate()
    if current_player == winner:
        return 1
    elif winner == empty:
        return 0
    else:
        return -1
```

get_move

嘗試跑幾輪，找到其中最多次 visit 的 action 當作答案。

```
def get_move(self, state):
    """
    Choose best move at given state
    """
    for _ in range(self.playout_num):
        state_copy = copy.deepcopy(state)
        self.play_once(state_copy)
    return max(self.root.children.items(), key=lambda child: child[1].visit_count)[0]
```

有了 MCTS 本體，接下來就可以製作 Agent。

把 MCTS 加進去，並且設定 playout_num 為 1000。

```
class Agent(object):
    def __init__(self, c=5, playout_num=1000, random=False):
        self.mcts = MCTS(c, playout_num)
        self.random = random
```

choose_action

最後是 test 階段會跑到的 choose_action。輸入會是一個 numpy array，輸出對應的 action。基本上就是去跟 MCTS 要 `get_move` 的結果。

```
def choose_action(self, state:np.array):
    start_player = state[0]
    board = Board(start_player)

    for i in range(1, 65):
        if state[i] != empty:
            board.states[i] = state[i]

    move = self.mcts.get_move(board)
    self.mcts.root = TreeNode(None, 5)
    return ' '.join(map(str, board.move_to_location(move)))
```

遊戲相關

遊戲相關的細節描述在 `Game` 與 `Board` 當中。

`Board` 可以描述一個盤面，也可以在上面給予 action 操作，基本上可以想像成遊戲本體。也包含了一些是否 terminal、是否出現贏家等等的判斷。大致上遊戲規則就寫在這個地方了

```python
def has_winner(self):
    # Build current board in numpy 3D array
    board = np.zeros((4, 4, 4))
    for move, player in self.states.items():
        # move, player = state
        location = self.move_to_location(move)
        board[location] = player

    # Check whether any player won the game
    for player in [player_O, player_X]:
        for i in range(4):
            for j in range(4):
                if sum(board[i, j, :]) == 4 * player \
                or sum(board[i, :, j]) == 4 * player \
                or sum(board[:, i, j]) == 4 * player:
                    return True, player
            if sum(np.diag(board[i])) == 4 * player \
                or sum(np.diag(np.fliplr(board[i]))) == 4 * player:
                return True, player
        if sum(board[i, i, i] for i in range(4)) == 4 * player \
            or sum(board[i, i, 3-i] for i in range(4)) == 4 * player \
            or sum(board[i, 3-i, i] for i in range(4)) == 4 * player \
            or sum(board[i, 3-i, 3-i] for i in range(4)) == 4 * player:
            return True, player
    return False, empty
```

而 `Game` 的目的在於可以實際上有兩個玩家去一起玩。這次這個部分只有在自己測試的時候會需要用到，其他時候就不需要了

```python
class Game(object):
    def __init__(self, board:Board):
        self.board = board

    def play(self, player1, player2, start_player):
        """
        We always regard player1 plays O
        While player2 plays X
        """
        while True:
            current_player = player1 if self.board.get_current_player() == player_O else player2
            move = current_player.choose_action(self.board)
            self.board.do_move(move)
            end, winner = self.board.is_terminate()
            if end:
                # print(f'Winner: {"O" if winner == player_O else "X"}')
                return winner
```

測試

如此一來我們就可以把盤面輸入，得到一個輸出了。目前在我的筆電 (CPU 為 intel i5-13500HX) 測起來大約需要花 2.8 秒的時間輸出。

```
if __name__ == '__main__':
    state = np.array([int(i) for i in open('../hw1-4_sample_input', 'r').read()[:-1].split(' ')])
    agent = Agent()
    agent.load_policy()
    agent.choose_action(state)
```

最後測試的結果，跟 Radom 相比 MCTS 都可以獲勝，然而跟同為 MCTS 的對手則先手會輸。