BE PROJECT REPORT

REPORT OF PROJECT ONE FOR EI339

Easy 21 Game

Gong Chen
Yang Zhaojing
Su Haodong
Department of Electronics and Telecommunications
SHANGHAI JIAO TONG UNIVERSITY

Contents

1 Introduction to Two Learning Methods: Q-Learning and Policy Iteration

1.1 Q-Learning

- **Summary**
  Q-learning is a **model-free** method which does not use the **transition probability distribution** and the reward function associated with the Markov decision process to learn or select action. Instead of learning utilities, Q-learning learns an **action-utility representation** $Q(s,a)$ which denotes the value of dong action a in state s.

- **Constraint Equation and Update equation**
  When the Q-values we calculated are correct, the constraint equation must hold at equilibrium:
  $$Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a') \tag{1}$$
  where γ is the discounting factor and $R(s)$ is the reward of state s.
  We can use this equation directly as an update equation for an iteration process that calculates exact Q-values. However, this requires a model also be learned because the equation uses $P(s'|s,a)$. If we don not want to learn about the environment, we can use the temporal-difference approach.

- **Update Equations in TD Approach**
  The temporal-difference(TD) approach requires no model of state transitions. The update equation is:
  $$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)) \tag{2}$$
  which is calculated whenever action a is executed in state s leading to state s'. α is the learning rate, γ is the discounting factor and $R(s)$ is the reward of state s.

- **Algorithm of Q-learning (off-policy TD control)**
  The pseudocode is showed in figure 1.

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

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop 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

Figure 1: Pseudocode of Q-learning

1.2 Policy Iteration

- **Algorithm**

  The policy iteration algorithm alternates the following two steps, begining from some initial policy π_0:

  Policy evaluation: given a policy π_i, calculate $U_i = U^{\pi_i}$, the utility of each state if π_i were to be executed.

  Policy improvement: Calculate a new MEU policy π_{i+1}, using one-step look-ahead based on U_i.

  The algorithm terminates when the policy improvement step yields no change in the utilities which is a solution to the Bellman equations. Because there are only finite many policies for a finite state space and each iteration can be shown to yield a better policy, policy iteration must terminate.

  The graphic explanation of the two steps of policy are show in figure 2.

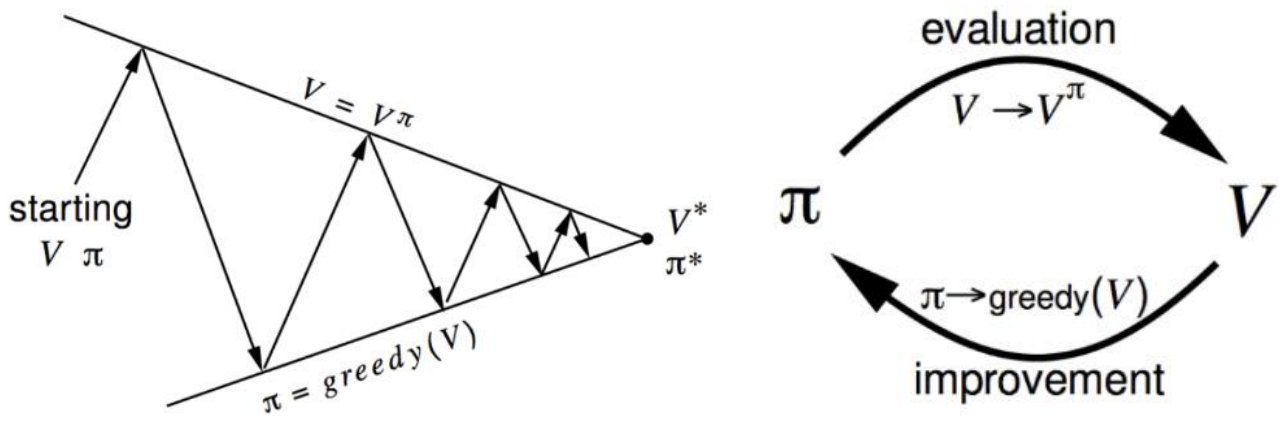

Figure 2: Policy Iteration

- **Pseudocode of Policy Iteration**

  The pseudocode of Policy Iteration is shown in figure 3.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 3: Policy Iteration

2 Implement of Two Learning Methods

2.1 Implement of Q-learning

2.1.1 Environment

The environment includes a class **State** and a function **step()**

- **State**: includes dealer's first card and player's current sum

```
    class State:
    dealercard = random.randint(1, 10) # initial value of dealer
    playersum = random.randint(1, 10) # current sum of the player
```

- **step()**: this function takes a state and an action(hit or stick) as inputs and returns a next state and a reward. In the easy 21 game we have two actions, **0 means to stick and 1 means to hit**. If we choose to stick, then the dealer will keep calling draw_card() function until going bust or achieving 16. And if the player choose to hit, then the player will call function **draw_card()**.

```
# action {0: stick, 1: hit}
# return state, reward
def step(state, action):
    if action == 1: # hit
        state.playersum = draw_card(state.playersum) # call function draw_card()
        if state.playersum > 21 or state.playersum < 1: # player goes bust
            return "terminal", -1.0
        else:
            return state, 0
    elif action == 0: # stick
        while(state.dealercard < 16):
            state.dealercard = draw_card(state.dealercard)
            if state.dealercard > 21 or state.dealercard < 1:
                return "terminal", 1.0
        if state.dealercard > state.playersum:
            return "terminal", -1.0
        elif state.dealercard < state.playersum:
            return "terminal", 1.0
        else:
            return "terminal", 0.0
def draw_card(current):
```

```
    card_num = random.randint(1, 10)
    if random.randint(1,3) < 3:
        current += card_num # black
    else:
        current -= card_num # red
    return current
```

2.1.2 Implement of one training episode of Q-learning

In one training episode, we initial a random state S. Then for each step of the episode, the agent use $\epsilon-greedy$ **policy**: choose action a randomly with probability ϵ and choose action a according to the policy derived from Q_table with probability $(1-\epsilon)$. After taking action, we observe reward $R(s)$ and state s' and update the Q_table using Bellman equation $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[R(s) + \gamma \max_a Q(s',a)]$

```
def q_learning(Q_table, epsilon, alpha, gamma, save_flag=1):
    state = State()
    # Initialize the state
    state.dealercard = random.randint(1, 10)
    state.playersum = random.randint(1, 10)
    while state != "terminal":
        action = None
        #epsilon-greedy policy
        if (random.random() < epsilon):
            action = random.randint(0, 1) # {0: stick, 1: hit}
        else:
            action = np.argmax(Q_table[:, state.dealercard, state.playersum], axis=0)
                # greedy
        old_dealercard = state.dealercard
        old_playersum = state.playersum
        state, reward = step(state, action) # take action and observe s', reward
        if state != "terminal":
            # Q(s,a) = Q(s,a) + alpha(r+gamma*Q(s',a')-Q(s,a))
            Q_table[action, old_dealercard, old_playersum] = (1-alpha) * Q_table[
                action, old_dealercard, old_playersum] \
            + alpha * ( reward + gamma*np.max(Q_table[:, state.dealercard, state.
                playersum]) )
        else:
            Q_table[action, old_dealercard, old_playersum] = (1-alpha) * Q_table[
                action, old_dealercard, old_playersum] \
```

```
            + alpha * reward
    return Q_table
```

2.2 Implement of Policy Iteration

2.2.1 Environment

The environment of policy iteration is the same as the environment of Q-learning.

2.2.2 Policy Evaluation

This function is used to evaluate the values of each state under the current policy using Bellman Expectation Equation:

$$V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))[r+\gamma V(s')] \tag{3}$$

In easy 21 game, there are two actions for a player's state: stick and hit. If the player chooses to hit, he has the probability to transit to another state or just going bust, and the value of this state under current policy can be calculated. However, if he chooses to stick, the state has to transit to another state but **there is no such state in state space standing for this action**. Thus, we have to calculate the value of each state in state space when choosing to stick. There are two methods to do this: **Sampling** and **Markov Stick-Value Matrix**.

- **Sampling**
  This naive way is just letting the player choose to stick at each state for a number of times and calculate the winning rate minus failing rate as the stick value of this state. The easy-understanding codes can be seen in the following codes:

```
def calculate_by_sampling(numOfgames):
    stick_value_table = np.zeros(shape=[3, 11, 22],dtype=float)
    # dealercard: 1-10   playersum: 1-21    0:draw   1:win   2:lose
    for i in range(1, 11): # dealercard: 1 - 10
        for j in range(1, 22): # playersum: 1 - 21
            for game in range(numOfgames):
                state = State()
                state.dealercard = i
                state.playersum = j
```

```
                _, reward = step(state=state, action=0)
                stick_value_table[int(reward), i, j] += 1 # number of winning,
                    losing or drawing
    stick_value_table = stick_value_table / numOfgames # number -> rate
    np.save('PolicyIterationData/stick_value_table_sampling.npy',
        stick_value_table)
```

- **Markov Stick-Value Matrix**

  Based on the Markov Stick-Value Matrix and initial values of some states, we use the transition updating equations to update the value of each state until all the values of the matrix converges. The size of the matrix M is $[22,22]$ and $M[i][j]$ means the value of the situation when the state is $dealer\_sum = i, player\_sum = j$ and it is dealer' turn to draw cards. According to the condition that dealer will stop drawing cards until reaching 16, we can assign initial values for $M[i][j]$ for $\forall i \geq 16, \forall j$:

$$M[i][j] \leftarrow \begin{cases} 1, & i \geq 16 > j \geq 1 \\ -1, & j > i \geq 16 \\ 0, & j = i \geq 16, \end{cases} \tag{4}$$

  The rough updating equations for the stick-value matrix can be seen as follows which does not consider going bust:

$$M[i][j] \leftarrow \sum_{m \in [1,10]} \frac{1}{15} M[i+m][j] + \frac{1}{30} M[i-m][j] \tag{5}$$

  We keep updating all the values of Markov Stick-Value Matrix until all the values converge. The code which considering situations of going bust can be seen as follows:

```
def calculate_by_Markov(max_iteration=2500):
    stick_value_table = np.zeros(shape=[22, 22]) # M[i][j], i=dealersum,j=
        playersum
    for i in range(16, 22): # Initialize
        for j in range(1, 22):
            if (i > j):
              stick_value_table[i][j] = -1
            elif (i < j):
              stick_value_table[i][j] = 1
            else:
```

```
                stick_value_table[i][j] = 0
    numOfiteration = 0
    while True:
        numOfiteration += 1
        for i in range(1, 16):
            for j in range(1, 22):
                new_value = 0.0
                for black_card in range(1, 11):
                    if (i + black_card > 21): # dealer goes bust and player wins
                        new_value += 1 / 15
                    else:
                        new_value += stick_value_table[i+black_card][j] / 15
                for red_card in range(1, 11):
                    if (i - red_card < 1):# dealer goes bust and player wins
                        new_value += 1 / 30
                    else:
                        new_value += stick_value_table[i-red_card][j] / 15
                stick_value_table[i][j] = new_value
        if numOfiteration > max_iteration:
            print("stick_value_table has been calculated successfully!")
            break
    np.save("PolicyIterationData/stick_value_table_Markov.npy",
        stick_value_table)
```

- **Comparison between Sampling and Markov Stick-Value Matrix**
  Through both theoretically analyzing and experimenting, we conclude that the latter method is far more fast than the former method. Before experimenting, we believe that the stick-values of each state calculated through two methods are the same, because the methods are essentially the same. However, when we print and draw the stick-values based on two methods, we are surprised to find that they are different. The graphic explanation are shown in figure 11.

After obtaining the stick-values of each state, we can use the Bellman Expectation Equation to evaluate the values of each state under current policy. The updating equation of each

state s under this easy 21 game situation is:

$$V(s) \leftarrow \begin{cases} stick-value-matrix(s), & \pi(s) = stick \\ \sum_{s'} p(s \rightarrow s')\gamma V(s'), & \pi(s) = hit \end{cases} \tag{6}$$

The policy evaluation function can be seen as follows:

```
def policy_evaluation(policy_table, stick_value_table, threshold, gamma):
    # dealercard: 1-10
    # playersum: 1-21
    value_table = np.zeros(shape=[11, 22])
    numOfiterations = 0
    print(stick_value_table.shape)
    for i in range(1, 11):
        for j in range(1, 22):
            if policy_table[i, j] == 0:
                value_table[i, j] = stick_value_table[1, i, j] - stick_value_table
                    [-1, i, j]  # value = win_rate - lose_rate
    while True:
        delta = 0
        numOfiterations += 1
        for i in range(1, 11):
            for j in range(1, 22):
                action = policy_table[i, j]
                if action == 0:
                    new_value = value_table[i, j]
                else:
                    new_value = action_1_value(i, j, value_table, gamma)
                delta = max(delta, abs(new_value-value_table[i, j]))
                value_table[i, j] = new_value
        print('value iteration %d: %.20f' % (numOfiterations, delta))
        if delta < threshold:
            print('Success!')
            break
    return value_table
```

2.2.3 Policy Improvement

We need to improve our policy based on the values of each state we calculated in Policy Evaluation according to:

$$\pi(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)[r+\gamma V(s')]$$
$$\Rightarrow \pi(s) = \begin{cases} 0, & stick-value(s) > \sum_{s'} p(s \to s'|\pi(s) = hit)\gamma V(s') \\ 1, & otherwise \end{cases} \tag{7}$$

The codes include two functions: action_1_value and policy_improvement. The former function is used to obtain **the value of choosing to hit** based on the value matrix calculated in policy estimation. The latter function is used to improve the policy. The codes can be seen as follows:

```
def policy_improvement(policy_table, value_table, stick_value_table, gamma):
    flag = True
    for i in range(1, 11):
        for j in range(1, 22):
            value_0 = stick_value_table[1, i, j] - stick_value_table[-1, i, j]# value
                =winning rate-losing rate
            value_1 = action_1_value(i, j, value_table, gamma)
            if value_0 > value_1:
                optimal_action = 0
            else:
                optimal_action = 1
            if optimal_action != policy_table[i, j]:
                flag = False
            policy_table[i, j] = optimal_action
    return flag
```

3 Results and Figures of Q-Learning

The winning rate we use is calculated through the following way:

For each initial state where player card and dealer card both range from 1 to 10, let the player use the policy(obtained from Q-table or policy matrix) he learned to play 1000 games and calculate the winning rate.

3.1 Learning Curve against episode number

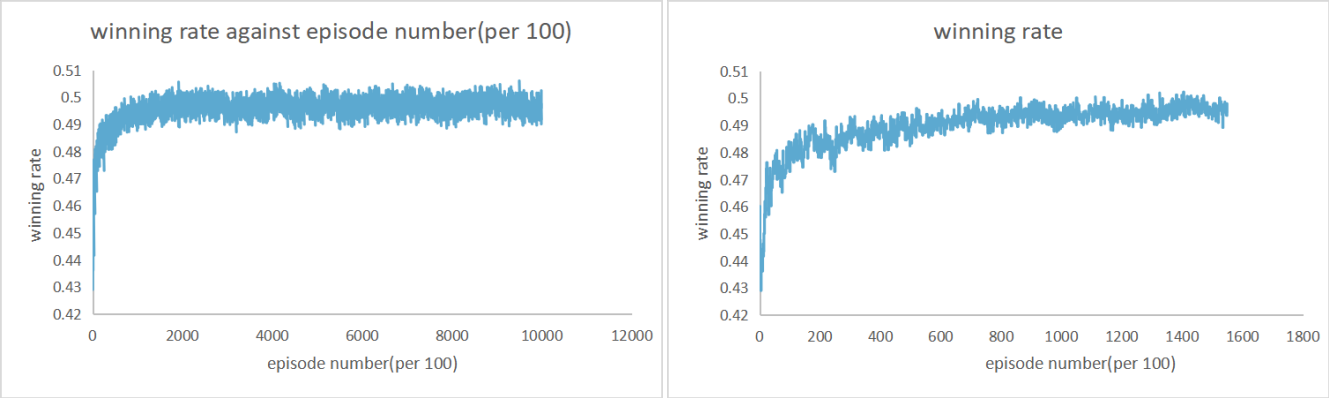

Figure 4: winning rate against episode numbers

The setting of the curve is: learning rate=0.01, discounting factor=1, exploration rate=0.3.

It can be show clearly from the left image of figure 4 that the winning rises from **0.42** at the beginning and converges to **0.5** after about 2000×10^2 episodes.

To see the increasing process more clearly, we draw another graph with episodes ranging from 0 to 1200×10^2, which directly shows that the **increasing rate** of winning rate is high at the beginning and then drops until the winning rate converges. The trend of the line is the same as we expected, because the Q-value table gradually converges and our policy of each state gradually converges.

3.2 Winning Rate with Different Learning Rate α

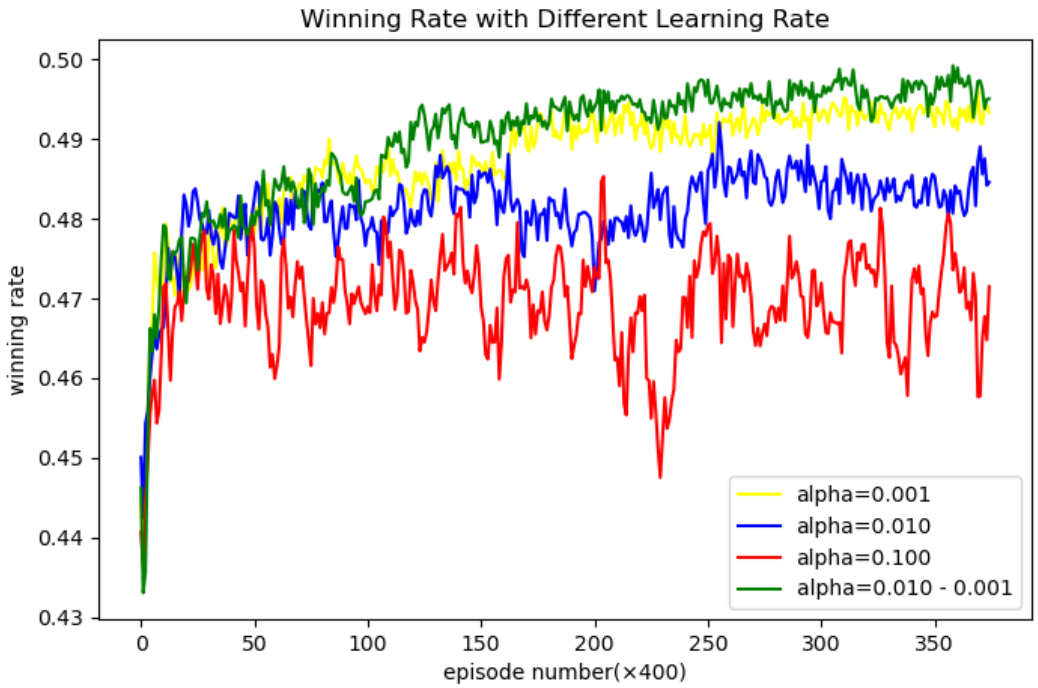

Figure 5: winning rate with different learning rate α

We have experimented **three fixed learning rate** $\alpha = 0.001, 0.01, 0.1$ and **one changeable learning rate** decreasing with the episode number from 0.01 to 0.001. Other hyperparameters are $\epsilon = 0.3$, $\gamma = 1.0$.The learning curve against episode number can be seen in figure 5 and we can obtain some conclusions:

- In the beginning 100×400 episodes, the the winning rate order and the increasing rate of winning rate order are the same: red > blue > green > yellow. This is the same as our theoretically analyzing, because the player with **more learning rate can update the Q-value table in a bigger step but also converges more quickly**.
- When all the players with different learning rate converge, we can directly see that the winning rate order is: green > yellow > blue > red. This is also in line with our expectation: **player with less learning rate can train the model more accurately and have a better result**.
- The player with **the largest learning rate can not converge** to a stable value even after a long training time, because a large learning rate causes the model to be easily

disturbed.

3.3 Winning Rate with Different Exploration Parameter ϵ

We have experimented four fixed exploration rate ϵ which is used in the ϵ-greedy policy. Other hyperparameters are $\alpha = 0.01$, $\gamma = 1.0$. The learning curve against episode number can be seen in figure 6.

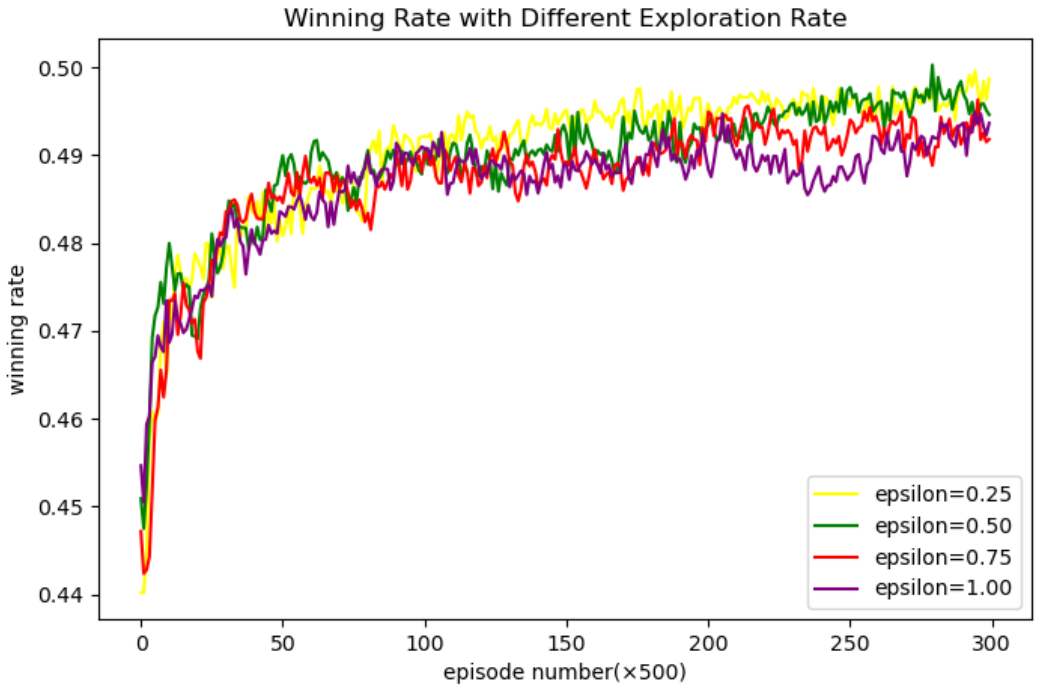

Figure 6: winning rate with exploration learning rate ϵ

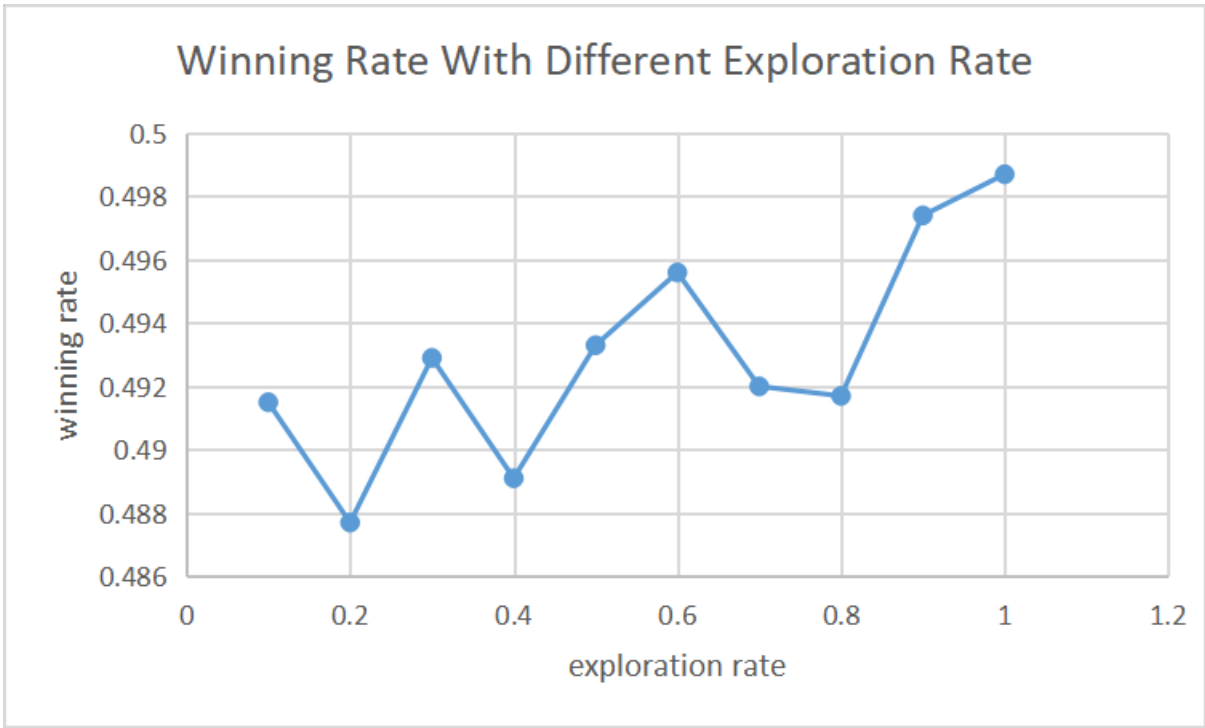

Figure 7: winning rate with exploration learning rate ϵ after 1000000 episodes

From the figure 6 and 7, We can directly obtain some conclusions:

- After about 250×400 episodes, the winning rate order is: yellow $\approx$ green > red > purple. The purple and red lines stand for $\epsilon = 1.00$ and $\epsilon = 0.75$, which are really large in the ϵ-greedy policy. This means that **we have a big probability to act randomly to explore more situations**, which is good for our final value table calculation. However, **the bigger exploration rate is, the harder it is for the model to converge**. As we can see, the red and purple line do not converge even after 300×500 episodes.

- Figure 7 is the winning rate after 1000000 training episodes where all the models with different exploration rate converge. We can see clearly that **player with the largest exploration rate win the most of the time**. This is the same as our expectation, because **he explores more situations and his Q-value table is more rational** by considering all the possible outcomes.

3.4 Optimal Policy of Each State

The optimal policy of each state can be seen in figure 8 where 0 means to stick and 1 means to hit.

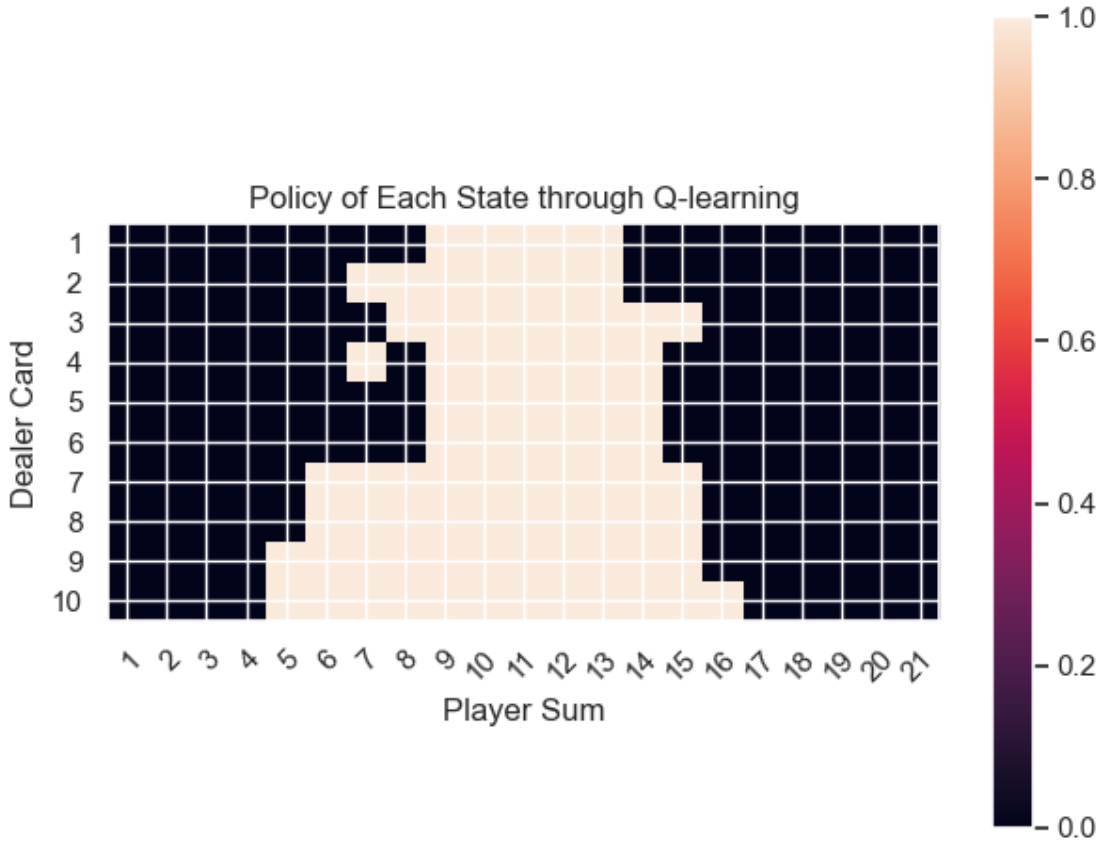

Figure 8: Policy of Each State of Q-learning

3.5 Optimal State-Value function

We show three 3D-graphs in this section which represent each state's value of choosing sticking, the value of choosing hitting and the optimal value. The graphs are show in figure 9 and figure 10.

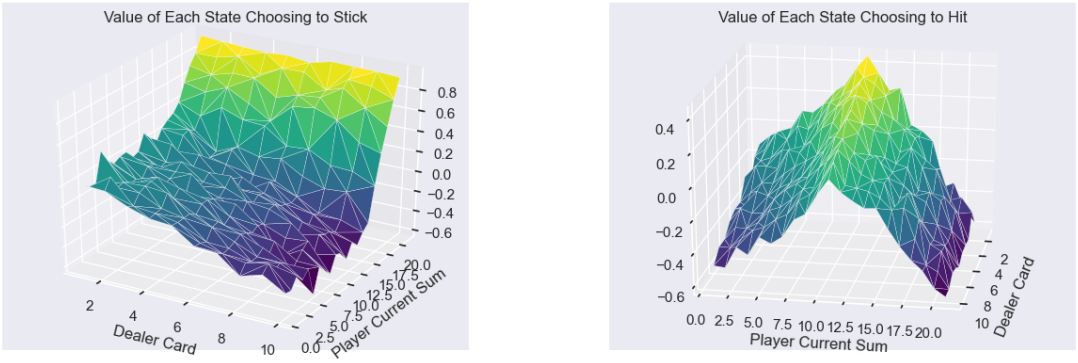

Figure 9: Value of Each State when Choosing Different action

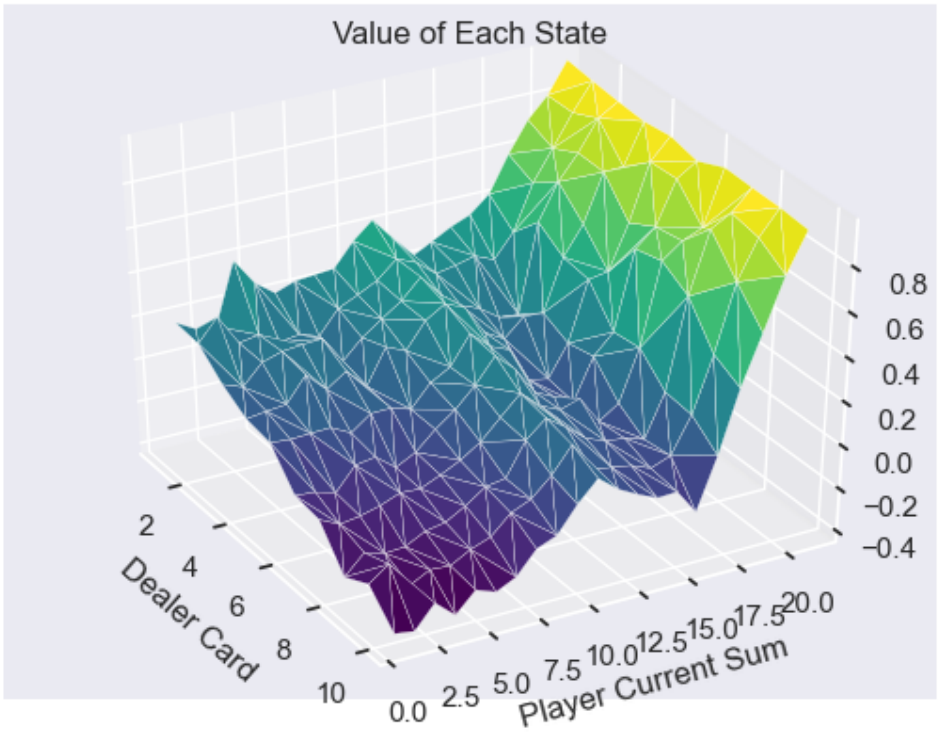

Figure 10: Value of Each State

4 Results and Figures of Policy Iteration

4.1 Comparison of Stick-Value Matrix through Different Methods

In the former section, we described our two methods of calculating the value of each state when choosing to stick. The resulting Stick-Value Matrix can be seen in figure 11.

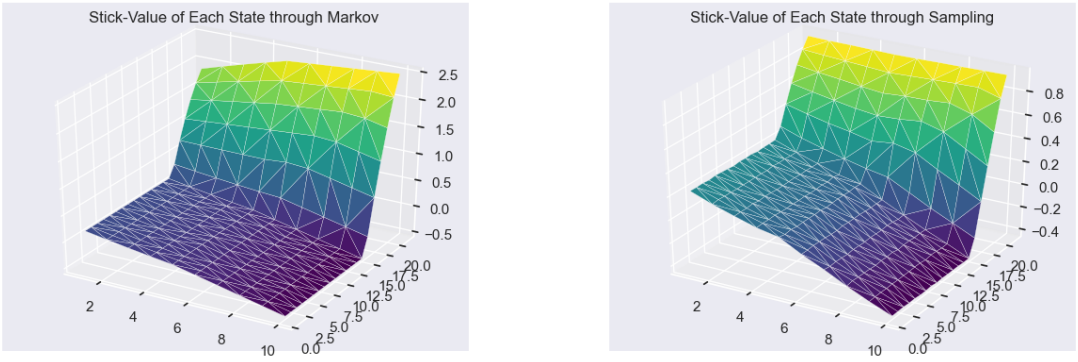

Figure 11: Stick-Value of Each State

Comparing the two graphs, we find that they are different when dealer's initial card is really small(from 1 to 5). In this situation, stick-values calculated through Markov are all relatively smaller than the stick-values calculated by sampling, which means that **Markov method is more timid and worries about the dealer getting larger sum in the future**.
Now, we can not decide which way is more accurate as well as reasonable and we will compare the final policies based on the two methods in next section.

4.2 Optimal Policy of Each State

The policies of each state calculated based on Markov and Sampling can be seen in figure 12 where 1 means to hit and 0 means to stick.

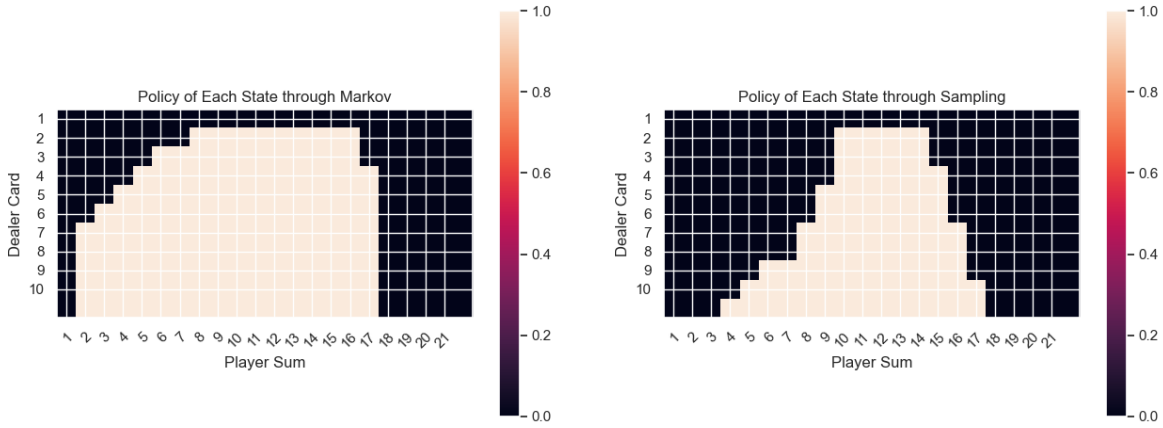

Figure 12: Policy of Each State through Policy Iteration

To understand which policy is better, we separately use two policies to play a large number of games with random starts and calculate the learning rates. The result is: the winning rate of **Sampling method** is **0.5102** and the winning rate of **Markov method** is **0.4677**.

4.3 Optimal State-Value function

Because we have two methods(Markov and Sampling) to calculate the stick-value of each state, we also have two value matrix of each state based on these two methods. They can be seen clearly through figure 13.

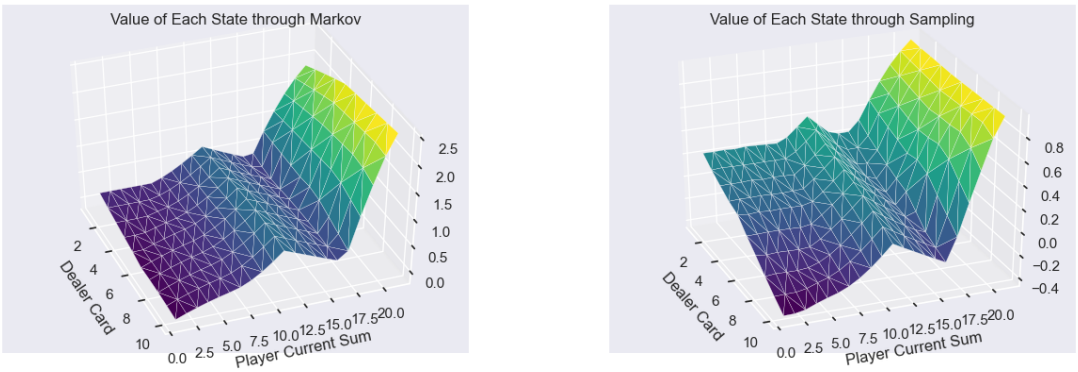

Figure 13: Value of Each State based on Policy Iteration

4.4 Winning Rate against Iteration Number

The learning curve against Policy Iteration number can be directly seen as follows:

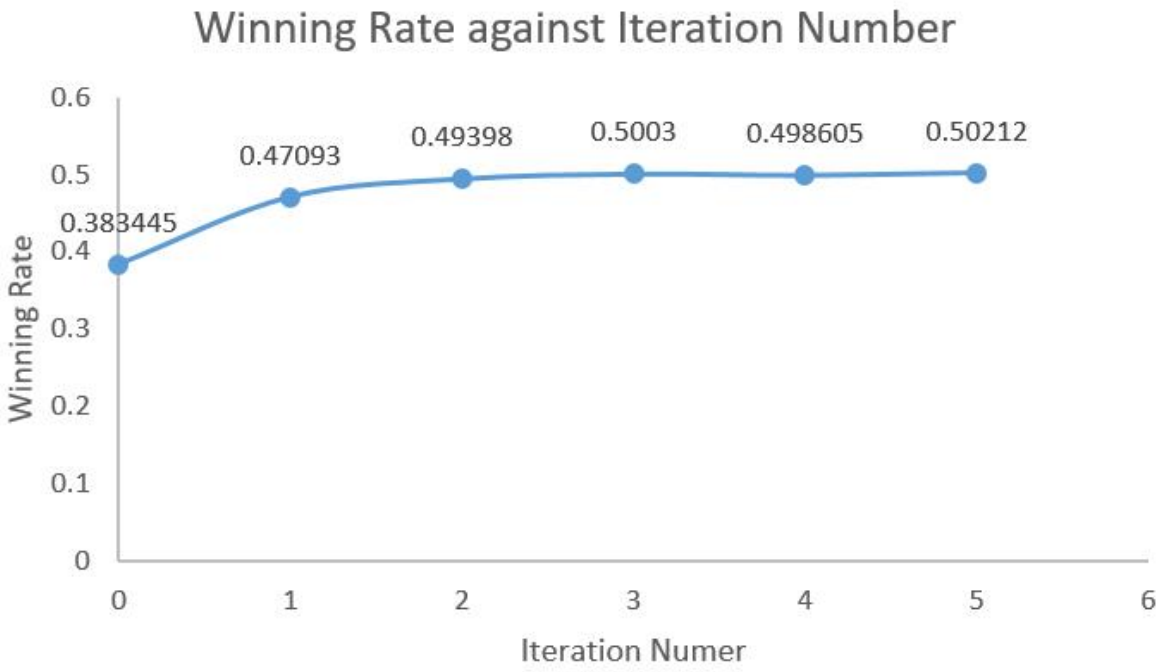

Figure 14: Winning Rate against Iteration Number

This shows that our model becomes better and more accurate with the increasing of policy iteration number.

5 Comparison between Q-learning and Policy Iteration

5.1 Performance Comparison

- **Effect Performance: Winning Rate**
  The maximum winning rate of Q-learning is **0.498** when $\alpha = 0.01$, $\gamma = 1$, training episodes=1000000 and the maximum winning rate of Policy Iteration is **0.5102** when using sampling method. **The effect performance of Policy Iteration is better**.

- **Practicing Performance: Time**
  The time spent on training Q-value table in Q-learning is really large. This is because when the hyperparameters like learning rate are not suitable, it takes a lot of time for the Q-value table to converge. Compared with Q-learning, **Policy Iteration is more time-saving** in easy 21 game. The reason is that the **number of actions is only 2** in easy 21 game and **the transition probability of states is also simple**. When updating the policy matrix, it is easier for the matrix to converge because it only has two values.

5.2 Characteristics Comparison

- Q-learning is an **off-policy and model-free** method. It does not need to learn about the environment like learning the transition probability distribution which is **practical in real life**. Also, it will choose the optimal policy according to the Q-value table it calculated and update the table per step, which **needs a long time to converge**.
- The biggest limit of Policy Iteration is that **it needs to know about the environment to evaluate and improve the policy**, which is impractical and impossible in real life. However, **if we know the environment, Policy Iteration is a better method** for reinforcement learning.