# Report of Introduction to Reinforcement Learning

Jie Liao 20740551

## I. Introduction

Q-learning and SARSA are algorithms that try to evaluate value function.

In Q-learning, the algorithm take actions based on policy derived from Q and update the Q value by taking the action that maximize Q value, which means the action I take in reality and the action I take to update the Q value might be different. The algorithm is shown in Fig.1.

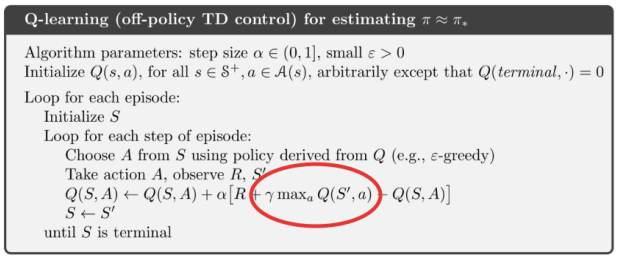



Fig. 1. Q-learning algorithm

In SARSA, the algorithm take actions based on policy derived from Q and update the Q value according to that action, which means the action I take in reality and the action I took to update the Q value is the same. The algorithm is shown in Fig.2.

The difference between these two algorithm is that Q-learning is an on-policy algorithm while SARSA is an off policy algorithm. The behavior policy and the evaluation policy are asynchronous in Q-learning while they are synchronous in SARSA. Besides, the Q value is updated by maximizing the Q value in Q-learning while it might not be the maximum Q value in SARSA. In general, Q-learning is more aggressive while SARSA is conservative.

The advantage of Q-learning is because of its aggressiveness, which makes it converge faster than SARSA. The disadvantage is that it will get less reward if the penalty of the 'cliff' is large after convergence because it's aggressive compared to SARSA.

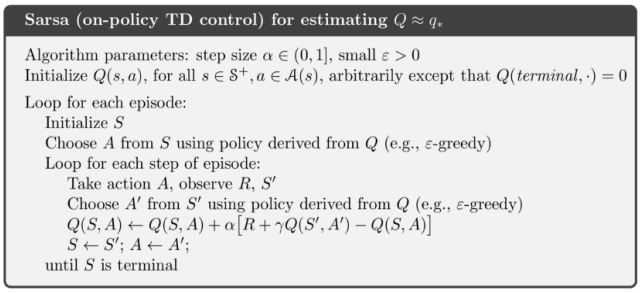



Fig. 2. SARSA algorithm

## II. Methods

The task of chess assignment is to learn an agent to checkmate a player that randomly make decisions. The crucial thing for the agent making the right move is to let the agent know the Q value of each step. Thus, I built a DQN to learn the Q value. A snippet of the network's architecture is in appendix A. I used two different algorithms in backpropagation, Q-learning and SARSA. The basic steps of these two algorithms are similar, the difference is how they compute and backpropagate the TD error.

### A. *Q-learning*

For Q-learning, the TD error is computed by taking the maximum Q value of allowed action. A snippet of the Q-learning's backpropagation is in appendix B.

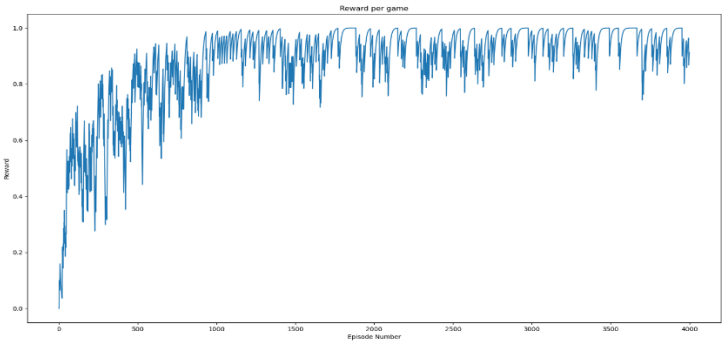
### B. *SARSA*

For SARSA, there is an additional step to choose the next action by epsilon greedy policy. After choosing the next action, the TD error is computed by taking the Q value of the chosen action.

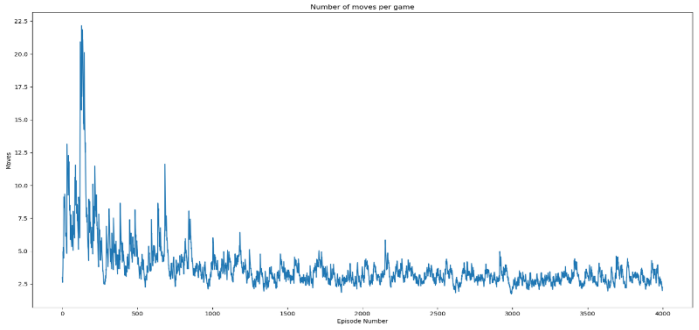
## III. Results

### A. *Comparision of Q-learning and SARSA*

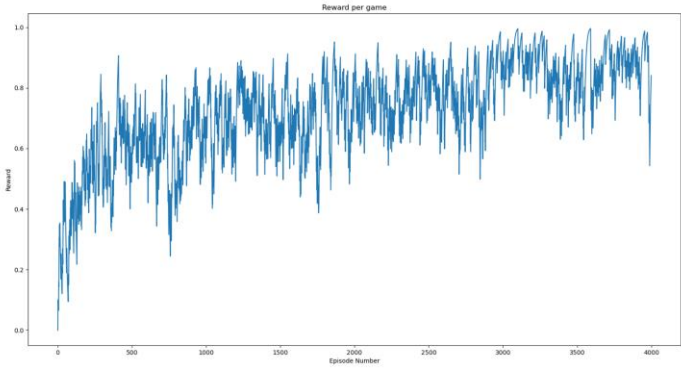
For analyzing the difference of Q-learning and SARSA, the discount factor γ and the speed β are set to be 0.8 and 0.00005. The results are shown in Fig. 3.



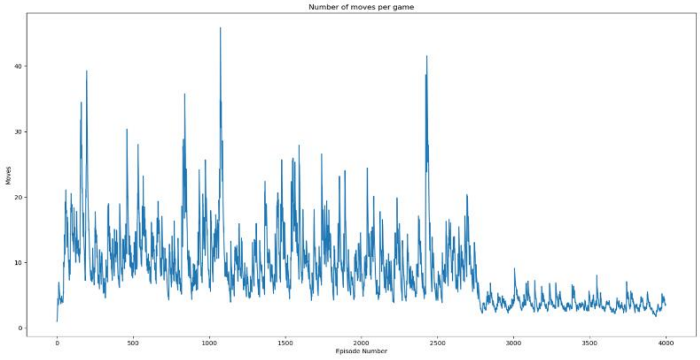

(a) Reward per episode of Q-learning




(b) Number of moves per episode of Q-learning




(c) Reward per episode of SARSA




(d) Number of moves per episode of SARSA

*Fig. 3. Comparison of results of Q-learning and SARSA algorithms. Figure (a) and (c) show the reward agent gets from each episode of game using Q-learning and SARSA. Figure (b) and (d) show the number of moves agent need to end the game using*

Comparing the reward and number of moves of Q-learning and SARSA, we can see that Q-learning converge faster than SARSA. Q-learning method needs approximately 1000 episodes to converge whereas SARSA needs about 3000 episodes. It is consistent with the introduction part that Q-learning is more aggressive than SARSA.

### *B. Impact of gamma*

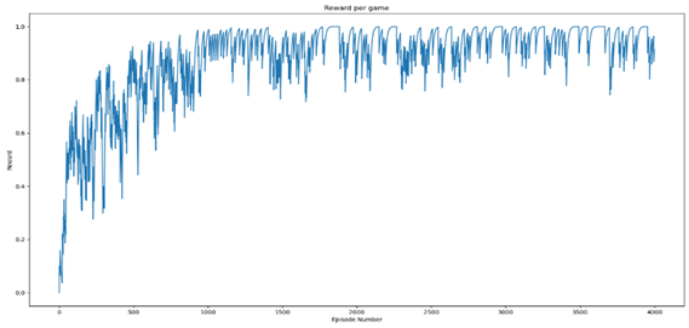
The experiment of gamma is conducted using Q-learning method and setting other parameters to be the same. Gamma is set to be 0.8, 0.9, 0.99. The results are shown in Fig. 4.

The plot shows that when gamma is increasing, the agent converge slower but oscillate slighter, which means the performance is better. To explain this, I would say when gamma is increasing, the agent focus more on the future, which is good for solving this task because agent will only get reward when they checkmate. On the other hand, it is more difficult for agent to learn when the agent focus more the future.

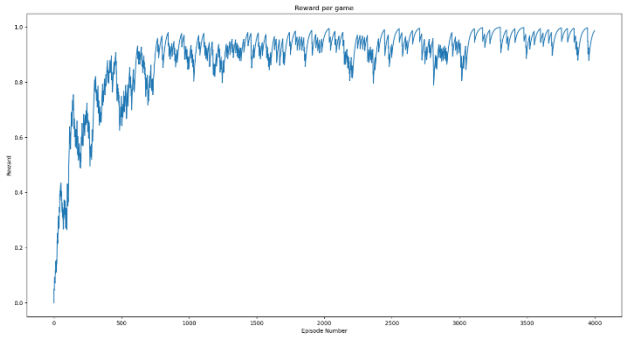
### *C. Impact of the speed β of the decaying trend*

The experiment of beta is conducted using Q-learning method and setting other parameters to be the same(with epsilon_0 = 0.2). Beta is set to be 0.005, 0.0005, 0.0005. The results are shown in Fig. 5.

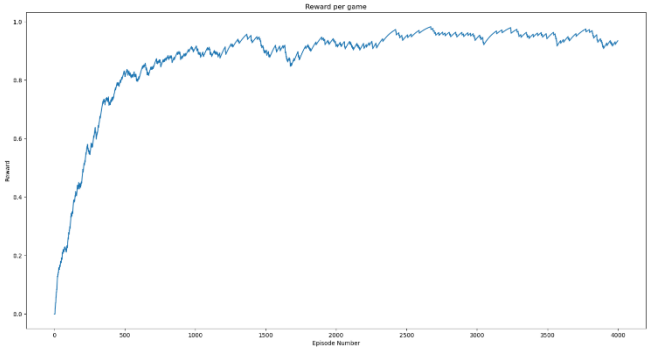
The plot shows that when beta is increasing from 0.0005 to 0.005, the agent's perform gets better and is more stable because agent's random actions become less when epsilon of epsilon greedy policy is decreasing faster.




(a) Reward per episode when gamma is 0.8

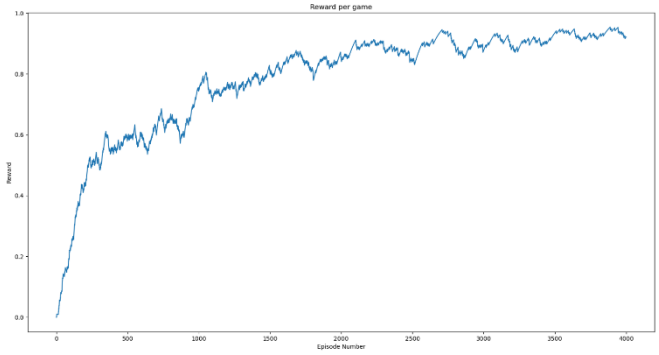



(b) Reward per episode when gamma is 0.95

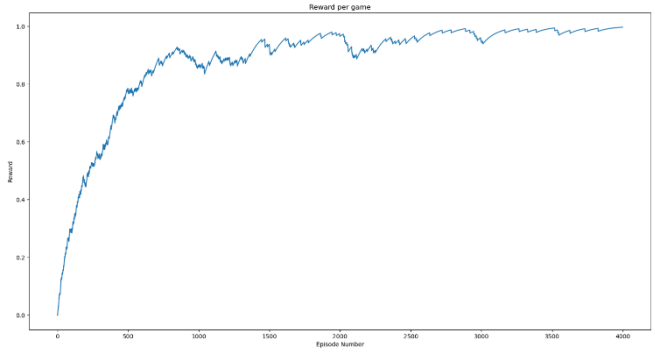



(c) Reward per episode when gamma is 0.99

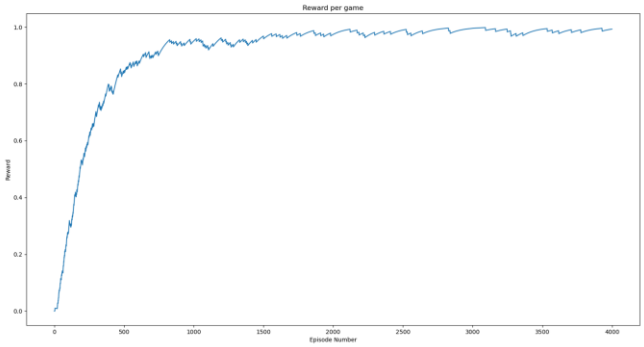
*Fig. 4. Comparison of results of different gamma value. Figure (a),(b)and (c) show the reward agent gets from each episode of game with different gamma.*




(a) Reward per episode when beta is 0.0005




(b) Reward per episode when beta is 0.005




(c) Reward per episode when beta is 0.05

*Fig. 5. Comparison of results of different gamma value. Figure (a),(b)and (c) show the reward agent gets from each episode of game with different beta.*

## IV. CONCLUSIONS

In general, this project builds a DQN to make agent learn to make decisions to play chess and experiment the difference of two algorithm, Q-learning and SARSA. Besides, the experiments with respect to value of gamma, beta show that the choice of hyperparameters can have large impact on the performance of the agent.

# Appendix

## A. *DQN network architecture*

```
import torch
import torch.nn as nn
import numpy as np
import torch.nn.functional as F

class Network(nn.Module):

    def __init__(self,n_input_layer,n_hidden_layer,n_output_layer):
        super(Network,self).__init__()
        self.hidden1 = nn.Linear(n_input_layer,n_hidden_layer)
        self.predict = nn.Linear(n_hidden_layer,n_output_layer)

    def initialize_weights(self):
        for m in self.modules():
            nn.init.kaiming_normal_(m.weight, mode='fan_in')

    def forward(self,x):
        out = self.hidden1(x)
        out = F.relu(out)
        out =self.predict(out)
        out = F.relu(out)
        return out
```

## B. *(a) Backpropagation algorithm of Q-learning(when game has not ended)*

```
if flag:
    # flag==1,Q-learning
    input_next = torch.tensor(X_next,requires_grad=True).float()
    output_next = network(input_next)
    Q_next =  np.copy(output_next.data.numpy())
    Qvalues_next=np.copy(Q_next[allowed_index_next])   # Q values of allowed actions
    Qmax_index_next = allowed_index_next[np.argmax(Qvalues_next)] # index of the maximum Q values in allowed actions

    target = torch.zeros_like(output)
    target[a_agent] = R + gamma*Q_next[Qmax_index_next]
    loss_weight = torch.zeros_like(output_next)
    loss_weight[a_agent] = 1.
    loss_weight = loss_weight.float()
    loss = loss_func(output,target) # (R+gamma*Q_next[Qmax_index_next]-output)²

    optimizer.zero_grad()
    loss.backward(loss_weight)
    optimizer.step()

    allowed_index = np.copy(allowed_index_next)
```

*(b) Backpropagation algorithm of SARSA(when game has not ended)*

```
else:
    # flag==0,SARSA
    input_next = torch.tensor(X_next,requires_grad=True).float()
    output_next = network(input_next)
    Q_next =  network.forward(input_next).data.numpy()
    Qvalues_next=np.copy(Q_next[allowed_index_next])
    a_agent_next = epsilon_greedy(epsilon_f,Qvalues_next,allowed_index_next,i)  # choose an action by epsilon greedy

    target = torch.zeros_like(output)
    target[a_agent] = R + gamma*Q_next[a_agent_next]
    loss_weight = torch.zeros_like(output_next)
    loss_weight[a_agent] = 1.
    loss_weight = loss_weight.float()
    loss = loss_func(output,target)  # (R+gamma*Q_next[a_agent_next]-output)²

    optimizer.zero_grad()
    loss.backward(loss_weight)
    optimizer.step()

    allowed_index = np.copy(allowed_index_next)
```

*(c) When game has not ended (the backpropagation algorithms of Q-learning and SARSA are the same)*

```
if Done==1:
    # if game has ended
    target = torch.zeros_like(output)
    target[a_agent] = R
    loss_weight = torch.zeros_like(output)
    loss_weight[a_agent] = 1
    loss = loss_func(output,target)    # (R-output)²

    optimizer.zero_grad()
    loss.backward(loss_weight)
    optimizer.step()

    R_save[n]=np.copy(R)
    N_moves_save[n]=np.copy(i)
    break
```