Rock, Paper, Scissors, a multi-agents approach, with Epsilon Greedy, Markov Chain, LSTM Cell and Actor Critic

Hanshan Li [*1] **Chongzhi Xu** [*1] **Zifan Li** [*1]

Abstract

"Rock paper scissors" is a simple and popular game, and it is possible for an agent to increase the winning rate by anticipating the opponent's next move through history records. Reinforcement Learning (RL) algorithms is applied in this project. Several computer players equipped with epsilon-greedy, epsilon greedy based Q-Learning, Markov Chain, LSTM based Double Deep Q Network and Actor Critic deep RL methods are set up respectively. First, competitions between computer players and human players indicate that both RL methods can enhance the winning rate(over 33%). Second, competitions between different computer agents are initialized, and Markov Chain is proved to be a better RL method over LSTM Cell, epsilon-greedy based Q-learning won over Actor Critic agent. Finally, the RPS game is upgraded into a more complex game named "Rock paper scissors lizard Spock" by adding two components, everything becomes more complex.

1. Introduction

"Rock paper scissors" (RPS) is an popular game integrated strategy and uncertainty. In this game, two player are supposed to choose one action from "rock", "paper" or "scissors" and show their choice in the same time. Statistically, if players play random choice, each player's probability of winning, tie, and losing is 1/3. Since each player has the same winning rate, the game is commonly used for two people for a random choice. In fact, the RPS game has only one Nash Equilibrium and it is optimal for each player to choose one of the three actions with the same probability 1/3. However, human beings do not play totally randomly for their action. A research shows that different people has different tendency in RPS games and humans are not good at choosing actions completely at random (1). That explains why some people seems to be an expert in RPS, since they can predict their opponents' choices based on history records. To make full use of history records and achieve better prediction, reinforcement learning (RL) is applied in RPS games. Methods came out in RPS competitions have shown that many different RL methods can achieve winning rate higher than 60%. However, since the the number of participants of RPS games increases, those methods with high winning rate are usually about the same, which indicates that different RL methods have their advantages and disadvantages.

To verify the advantages and disadvantages of different RL methods, epsilon-greedy, epsilon greedy based Q-Learning, Markov Chain, LSTM based Double Deep Q Network and Actor Critic method were implemented respectively for computer RPS players in this paper. Competitions between computer players with the two methods and human players were carried out respectively and the results showed that all methods can have a improved winning rate when competing with human players. Then, competitions between computer players under two methods were carried out. By initializing 1000 rounds of play, the result showed that Markov Chain is the better RL method. Later, the RPS game was expanded into a 5-action game named "rock paper scissors lizard spock", and similar results are achieved based on the same two RL methods.

The paper is organized as follows. Section 2 provides background material. Section 3 introduces the way the Q-Learning and the Markov Chain are used in RPS. Section 4 provides results and analysis for competitions for both RPS and "rock paper scissors lizard spock". Section 5 provides the conclusions.

2. Background

All submissions must follow the specified format.

2.1. ϵ-Greedy

As the book (2) shows, ϵ-greedy is a straight forward algorithm that can balance exploration and expectation. In

[*]Equal contribution [1]Department of Electrical Engineering, Columbia University, New York City, New York, USA. Correspondence to: Hanshan Li <hl3515@columbia.edu>, Chongzhi Xu <cx2273@columbia.edu>, Zifan Li <zl3095@columbia.edu>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021. Copyright 2021 by the author(s).

Algorithm 1 Q-Learning Algorithm

Input: $Q(S, A)$ and $Q(\text{terminal state}, \cdot) = 0$
Output: (near-) optimal policy $\pi*$
Repeat: for each episode
Initialize S
repeat
Choose action A from state S following the policy π derived by Q (e.g., ϵ-greedy policy)
Take action A, obtain reward R and the next state S'
Update Q value as
$Q(S, A) \leftarrow Q(S, A)+$
$\alpha\left(R+\gamma \max_a Q\left(S', a\right)-Q(S, A)\right)$
$S \leftarrow S'$
until S is the terminal state

the ϵ-greedy method, the agent explores with probability ϵ and exploits with probability (1 - ϵ). In an environment with *k* action choices, the probability of picking the j^{th} arm, $Pr(A_t = j)$, is distributed according to equation (1).

$$\begin{aligned}\Pr\left(A_t=j\right) &= \begin{cases}1-\epsilon+\frac{\epsilon}{k} & \text{if } j=\underset{j=1,\ldots,k}{\arg\max}\mu_j, \\ \frac{\epsilon}{k} & \text{otherwise.}\end{cases} \\ &= \begin{cases}1-\frac{(k-1)\epsilon}{k} & \text{if } j=\underset{j=1,\ldots,k}{\arg\max}\mu_j, \\ \frac{\epsilon}{k} & \text{otherwise.}\end{cases}\end{aligned} \tag{1}$$

The pseudo code of ϵ-greedy algorithm is presented in Algorithm 1. To balance the exploration or exploitation, the ϵ is needed to be chosen appropriately. If ϵ is close to 1, the algorithm would run with zero exploitation. On the other hand, if ϵ is down to 0, the algorithm would become greedy policy.

2.2. Q-Learning

Q-Learning is an off-policy control method, which is summarized in algorithm 2. In Q-Learning, state value is updated at time step t based on the instantaneous reward and the estimated return on the next state. In algorithm 2, action A is chosen following the ϵ-greedy policy, which is the same as will discussed in section 3.

2.3. Markov Chain

Markov Chain is a stochastic process with Markov property that exists in discrete index set and state space in probability theory and mathematical statistics (3). It is defined by transfer matrices and transfer diagrams. Markov Chain can be also described as Markov process, which is a memory-less random process. When rewards and decisions are implemented, Markov process can be described as Markov decision process (MDP). Besides, Markov Chain can be also used in describing Monte Carlo method.

2.4. Deep Q-Network (DQN)

Naive Q-Learning usually has stability issues that will oscillates or diverges with neural nets. Thus, DQN was introduced and could provide a stable solution to deep value-based RL. DQN utilizes two primary techniques: *fixed target Q-network* and *experience replay*. The first technique uses an older set of the parameters θ^- for the Q-Learning target y_i, which is shown in equation (2).

$$y_i = r + \gamma \max_{a'} \hat{Q}\left(s', a'; \theta^-\right) \tag{2}$$

Using the *fixed target Q-network*, the loss function can be rewritten as equation (3)

$$\mathcal{L}_i\left(\theta_i\right) = E_{s,a,r,s' \sim \rho}\left[\left(r + \gamma \max_{a'} \hat{Q}\left(s', a'; \theta^-\right) - Q\left(s, a; \theta_i\right)\right)^2\right] \tag{3}$$

Then, to optimize the weights θ_i of the deep neural network at iteration i, the loss function can be differentiate with respect to θ. Equation (4) is shown as follows.

$$\begin{aligned}&\nabla_{\theta_i}\mathcal{L}_i\left(\theta_i\right) = \\ &E_{s,a,r,s' \sim \rho}\left[\left(r + \gamma \max_{a'} \hat{Q}\left(s', a'; \theta^-\right) - Q\left(s, a; \theta_i\right)\right) \nabla_{\theta_i} Q\left(s, a; \theta_i\right)\right]\end{aligned} \tag{4}$$

Using the method of stochastic gradient descent (SGD), the expectation in $\nabla_{\theta_i}\mathcal{L}_i\left(\theta_i\right)$ is replaced by one sampled value and the neural network weights θ can be updated as equation (5).

$$\theta_{i+1} = \theta_i + \alpha \nabla_{\theta_i}\mathcal{L}_i\left(\theta_i\right) \tag{5}$$

Another novel technique *experience replay* builds a dataset from the experiences taht the agent has seen at each time step. It learns more efficiently from data and reduces the variance in the Q-Learning update. Furthermore, the random sampling in experience play diminishes correlations which exist in traditional Q-Learning.

The pseudo code of DQN is presented in algorithm 3. ϵ-greedy method is applied in for the agent to choose actions. A function ϕ is also implemented for a fixed length representation of histories, which is fed as inputs to the neural network

2.5. DDQN

Double Deep Q-Learning (DDQN) is an improvement from DQN. The optimal choice of action in DQN is the parameter

Algorithm 2 DQN Algorithm

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode = 1, 2, . . . , M **do**
Initialize sequence $s_1 = x_1$ (x_i is the raw data from the environment) and preprocess the sequence $\phi_1 = \phi(s_1)$
Execute action a_t and observe reward r_t and the new data input $x_t + 1$ from the environment
Set $s_t + 1 = s_t, a_t, x_t + 1$ and preprocess $\phi_t + 1 = \phi(s_t + 1)$
Store transition $(\phi_t, a_t, r_t, \phi_t + 1)$ in D
Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_j + 1)$ from D
Set $y_j = \begin{cases} r_j, \text{ if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right), \text{otherwise} \end{cases}$
Preform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters $theta$, i.e., update in (5)
Every C steps reset $\hat{Q} = Q$
end for

Algorithm 3 Actor-Critic Algorithm

Input: A differentiable policy parameterization $\pi(a|s, w)$ and a differentiable state-value parameterization $\hat{V}(s, \theta)$, step sizes α and β
Output: (near-) optimal policy $\pi * (\cdot|\cdot, w)$
Initialize: Initialize policy weights θ and state-value weights w
Repeat:
Initialize first state of episode S
Assign initial value 1 to I
While S is not terminal:
1. $A \sim \pi(\cdot|S, \theta)$
2. Take action A, observe S', R
3. $\delta \leftarrow R + \gamma \hat{V}\left(S', \theta\right) - \hat{V}(S, \theta)$ $\left(\hat{V}\left(S', \theta\right) = 0 \text{ if } S' \text{ is terminal }\right)$
4. $\theta \leftarrow \theta + \beta \delta \nabla_w \hat{V}(S, \theta)$
5. $w \leftarrow w + \alpha I \delta \nabla_w \log \pi(A \mid S, w)$
6. $I \leftarrow \gamma I$
7. $S \leftarrow S'$

θ_t of the target-Q network, which will cause overestimation of Q value. To avoid this overestimation, the optimal choice of action in DDQN is based on the parameter θ_t of the Q network which is currently being updated. The target function of DDQN is shown in Equation (6).

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q\left(S_{t+1}, \underset{a}{\operatorname{argmax}} Q\left(S_{t+1}, a; \boldsymbol{\theta}_t\right), \boldsymbol{\theta}_t^-\right) \tag{6}$$

2.6. Actor-Critic

Actor-Critic is an approach to reduce variance caused by Monte Carlo sampling. In this method, an *actor* is applied to update the policy parameters w, while a *critic* is applied above the actor to update the state value function parameter θ to evaluate the current policy, and suggest the actor to improve its policy. Pseudo code for Actor-Critic algorithm is presented in Algorithm 4.

2.7. LSTM

Long Short-Term Memory (LSTM) is a special kind of artificial Recurrent Neural Network (RNN) architecture, which is widely used in Deep Learning. In the LSTM model, two distinct states passed between the neurons, which are called the cell state and the hidden layer. The cell state is used for storing short term memory, while the hidden layer, on the other hand, focuses on long term memory. The LSTM model can be further separated into three vectors: a forget vector, an input vector, and an output vector. These vectors choose which information is needed to drop.

3. Approach

Leveraging existing algorithms and pseudo code discussed in Section 2 , we managed to implement 5 different computer agents. Section 3 and Section 4 below are respectively the explicit approaches for ideas, structures and the experiment results from pairwise confrontations between them.

Basically, for the sake of convenient cooperation, we created a Google Colaboratory file as the platform for agents to be called and to fight with each other. What's more, we also developed two interactive cells for lab viewers, one is for playing Rock Paper Scissors and one is for experiencing Rock Paper Scissors Lizard Spock, you are able to choose any of our agents to play with and see the astonishing power inside the agents!

As we designed different computer agents, we ourselves will play with each agent for 50 rounds to test their ability to deal with human strategy. In the meanwhile, the algorithm will play a hundred rounds with other designed programs. The algorithm with the higher number of rounds won takes the match, by which we can judge the performance of an algorithm.

It is worthy to mention that since both Deep Q Network strategy agent and Actor Critic strategy agent take a deep neural network to play their steps, it would take an unbearably long time to let them play for several thousand epochs considering the limited computational resources on Google Colab. Therefore to keep a balance between the exploration

of experimental results and time saving is quite important for us.

3.1. Simple ϵ-Greedy Strategy Agent Approach

As we mentioned in the Section 2.1, in our implementation, a simple ϵ-Greedy Strategy Agent is defined to simply play the same actions if it won in the previous step with 1-ϵ percentage chance , in other words, do exploiting. While if it happens to be the other ϵ chance or it lost in the previous round in the competition, it will play randomly as exploration. Without a memory embedded, this simple ϵ-Greedy Strategy agent is only learning from the previous one round result, it is just a demonstration on how a agent could be build since an agent could easily be countered is definitely not we desire.

3.2. ϵ-Greedy Strategy based on Q-Learning Update Agent Approach

In order to go one step further than just simply ϵ-Greedy Strategy, we added a "memory" inside the agent with a format of Q-matrix with 3 rows and 3 columns respetively representing the opponent play and the agent's own play, namely Rock, Paper or Scissors. The Q-matrix keeps updated for every game round, with the opponent's play before previous play as **state**, with its own play before previous play as an **action** and with opponent's previous play as **next state**. **Reward** is set by leveraging a helper function Get_Reward, so that agent gets encouraged when it plays right in the previous round, otherwise gets punished when it loses or is tied with its opponent in the previous round. Finally we implement our agent learning process by updating the Q value to the element, row [state] column [action] in Q-matrix according to the Equation in Algorithm mentioned in Section 2.

In each round, the agent is asked to offer a new move Rock, Paper or Scissors without being given the rival's move, then it is granted a chance to learn from the result of the match. At the beginning of the competition, or more explicitly the first 2 rounds, our ϵ-Greedy Strategy based on Q-Learning Update Agent is lack of experience thus is allowed to play randomly, and in the meantime initialize a Q-matrix based on the game result. Starting from the third round, our agent play its move with ϵ-greedy on top of the Q-matrix, that is to say, playing greedily with a chance of 1-ϵ probability to exploit by giving the optimal action at the state on Q-matrix, while playing randomly by a probability of ϵ to explore.

For instance, suppose the agent's rival is an always-Rock player, results from the first two matches are Rock by opponent, Paper by agent, then Rock by opponent, Scissors by agent. In this situation at the initial period all nine elements of the Q-matrix are zero, then the upper-left corner of agent Q-matrix will be updated to a plus number indicating that it's right to play Paper to cover Rock. On the contrary, the upper-right corner of agent Q-matrix will be updated to a minus number indicating that it's awful to play Scissors to challenge Rock.

3.3. LSTM based on Deep Q Network Strategy Agent Approach

The objective of this method is to construct an AI agent to play rock-paper-scissors games with humans or other algorithms using reinforcement learning (RL) technique - DDQN algorithm. When it comes to deep reinforcement learning, we like to see that it demonstrates some ability to counter the changing strategy and generate a better win rate than the opponents. Considering the rock paper scissors environment, the LSTM cell is the first method that fits the environment, because the LSTM is good at handling time or step sequence input, just compatible with game round setting. But only LSTM cells can just learn a sequence with fixed pattern and requires a large training dataset. So, DDQN is selected. The use of max can quickly make the Q-value close to the possible optimization target, and DDQN achieves the elimination of the overestimation problem by decoupling the two steps of target Q-value action selection and target Q-value calculation. Based on the information above, we design the environment as shown in the following Fig. 1.

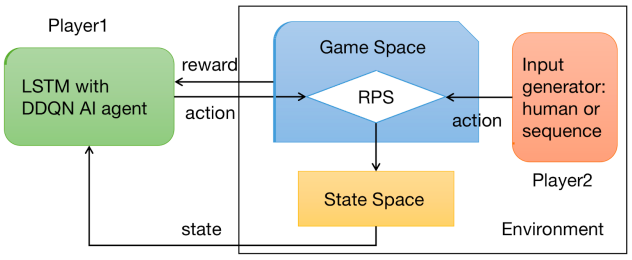

Figure 1. Environment designed for LSTM method.

Fig. 2 is the double DQN model diagram of the pseudo code. In the overall architecture of the dual DQN agent design, the yellow part is coded inside the stepwise approach and iterated with an empirical replay batch size of N samples. The green part is the design of the control exploration and development actions. The orange part is the main policy-based action model. The weights of the action models are periodically transferred and discounted to the target model.

In this project, we use the most common LSTM cell structure that has sequential input and single output, the structure is shown in Fig. 3. We will give the model 2 kinds of input, human choice and a fixed sequence of RPS to test its ability. The sequence length is a fixed length that suffices the requirement of numbers of LSTM input. Results were

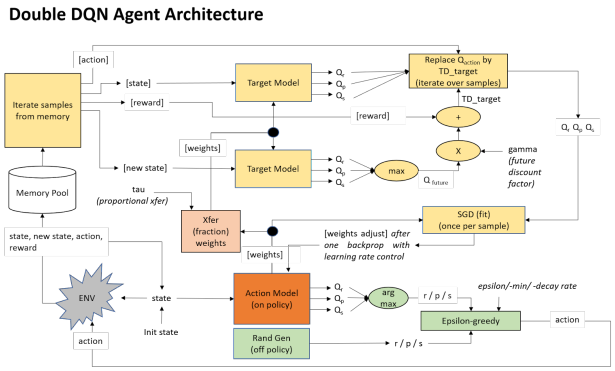

Figure 2. Double DQN model agent architecture.

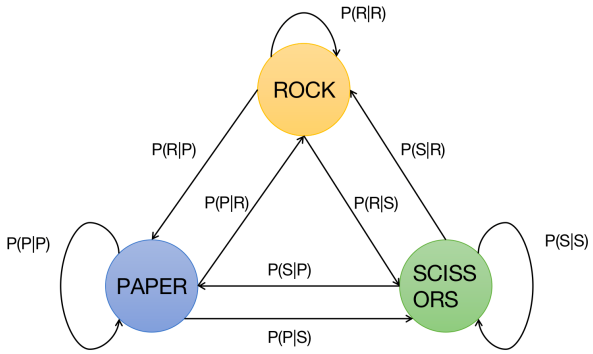

Figure 4. Rock Paper Scissors state transition .

shown in section 4.

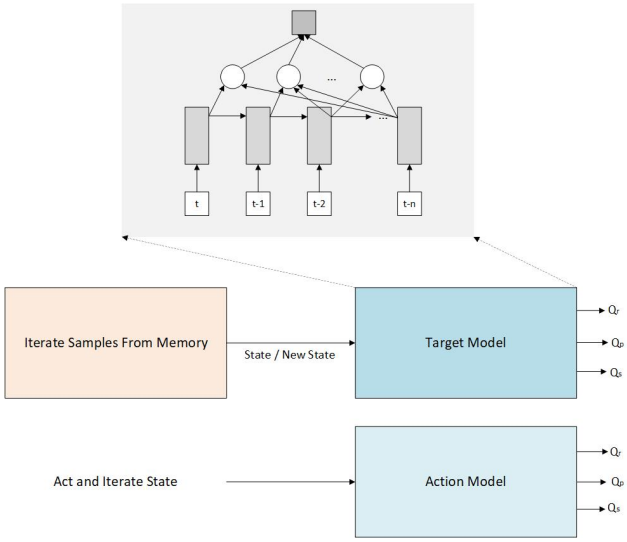

Figure 3. LSTM cell model architecture.

3.4. Markov Chain model Approach

This approach requires a discrete Markov Chain model to be implemented, which is very straightforward. We can easily define the Markov chain in a rock paper scissors environment, as shown in Fig. 4 The main problem for us is how to use this simple state transition topology. A natural idea is to analyze the input and the output from the latest round and try to predict the next input and give the next move. So, the current state should be a pair in the form of (output, input) and the next state is Markov's output. But this prediction method always suffers from the limitation in long run games, because the history record does make a difference. To solve this problem, we need to set up a memory parameter and a history list. History list will record the past several rounds results and memory decides how many of them the model should remain. Now we can start to design our strategy of the "next move". In fact, there are plenty of strategies in the Rock Paper Scissors game, but all methods basically work based on changing the probability of actions in the state transition matrix.

Here we implemented 2 manual strategies called input/output oriented and random protection. To be more specific, input and output oriented algorithms can be described as based on the Markov state transition choice prediction, output the option rivals our prediction or the prediction itself. Random protection is used when the opponent keeps winning and the Markov chain keeps losing or the accumulated score(reward) is incredibly low, so randomly output rock paper scissors for 5 rounds to refill the history list and attempt to find an opportunity to fight back. To conclude this part, the Markov chain model works in 3 steps: 1) gives the prediction by using a history list to update the state transition matrix 2) then selecting what strategy (input or output oriented and random protection) to use 3) compare with the input and record it in the history.

3.5. Actor Critic Strategy Agent Approach

As what we discussed in Section 2.5, the Actor Critic algorithm exists to reduce the variance using a critic, it is able to accept infinite input and infinite output and handle a complicated job. The actor takes in the current environment state and determines the best action to take, while the critic plays the evaluation role by taking in the environment state and action and returning an action score. The actor is like a child playing on the playground, keeps selecting desired actions and moving from one place to another, while the critic is more like a parent to look after the child and guide the child to do the right thing.

Both the actor and critic contains a set of parameters, for

actor is the policy parameters θ suggested by critic, and for critic is the action-value function parameters w. In our code design we embed these two set of parameters in custom neural networks, every node inside the neural networks acts as a parameter element, it will be trained through each step the agent makes, and gradually as the neural network converges, the two set of parameters will also become optimal for our agent, indicating that our Actor Critic Strategy agent is already adapted to its rival. The below mind mapping illustrates the rest idea of our code design is shown in Fig. 5. It's worth noting the learning rate for actor and critic are

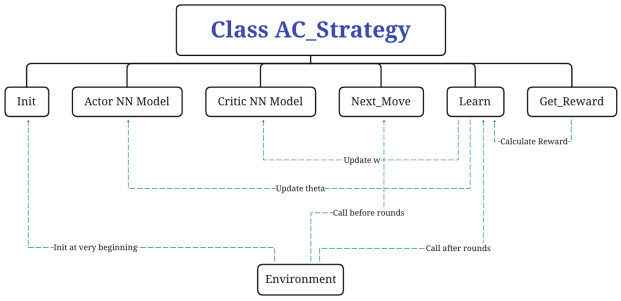

Figure 5. Actor Critic Structure for Rock Paper Scissors.

entirely different and should be given respectively alpha and beta when initialization. Careful attention should be paid to the hyperparameters setting, including alpha and beta mentioned above, gamma and rewards, for the AC algorithm agent is extremely sensitive to them.

4. Experiment Results

4.1. Before Competition

One key consideration before we start the experiment is to ensure all our agents are actually learning instead of playing irregularly without rules. Like we mentioned in Section 3.2, we composed a "cute" player that always plays Rock. If an algorithm can't perform overwhelming triumph over the always-Rock player, then all the complexity inside will turn out to be useless and meaningless.

As expected, the scales of victory always tip in all of our agents' favour without doubt, the win rate of all algorithms are extremely high over the always-Rock player. But for the sake of the upper limit of pages, we won't attach results here, you are able to check in the Google Colab Environment v.s. Environment part by selecting Strategy1 as always-Rock player and select Strategy as the algorithm you want to check.

4.2. Simple ϵ-Greedy Strategy Agent v.s. ϵ-Greedy Strategy based on Q-Learning Update Agent

To start a competition, the Strategy and Strategy2 as contestants are required to be chosen, and carefully given parameters. Take ϵ-Greedy Strategy based on Q-Learning Update agent for example, it contains 3 parameters used for initializing the object, respectively epsilon, min_eps, epoch_to_min_eps as we discussed in Section 3.2. As Fig.7

```
#-------------------------------------------------------------------------------
# Assign episodes to ends. Change this!!!
Max_episodes = 300
episodes = 1
verbose = 1
#-------------------------------------------------------------------------------
# Choose your Strategy1 from below by uncommend!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
#Strategy1 = Simple_Strategy()
Strategy1 = Epsilon_Strategy()
#Strategy1 = Epsilon_Q_Learning_Strategy(epsilon=0.5,min_eps=0.2,episodes_to_min_eps=Max_episodes)

# Choose your Strategy2 from below by uncommend!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
#Strategy1 = Simple_Strategy()
#Strategy2 = Epsilon_Strategy()
Strategy2 = Epsilon_Q_Learning_Strategy(epsilon=0.5,min_eps=0.2,episodes_to_min_eps=Max_episodes)
#Strategy2 = AC_Strategy(alpha=0.001,beta=0.001)

#-------------------------------------------------------------------------------
```

Figure 6. Contestants' Parameters Settings.

shows, Simple ϵ-Greedy Strategy agent is completely defeated by ϵ-Greedy Strategy based on Q-Learning Update agent. The main reason for the result lies in the Q-matrix embedded in the latter. The Q-matrix enables the ϵ-Greedy Strategy based on Q-Learning Update agent to learn on top of the Q-matrix updated from every single history play pairs, which offer a state-action guide in the whole process. However, without a memory to rely on, the Simple ϵ-Greedy Strategy agent can only make a reckless decision sorely based on the previous round result.

```
290th round, Strategy1: P, Strategy2: S, Strategy2 is winner, now scores are 107:183
291th round, Strategy1: R, Strategy2: S, Strategy1 is winner, now scores are 108:183
292th round, Strategy1: R, Strategy2: P, Strategy2 is winner, now scores are 108:184
293th round, Strategy1: R, Strategy2: P, Strategy2 is winner, now scores are 108:185
294th round, Strategy1: R, Strategy2: P, Strategy2 is winner, now scores are 108:186
295th round, Strategy1: S, Strategy2: P, Strategy1 is winner, now scores are 109:186
296th round, Strategy1: P, Strategy2: S, Strategy2 is winner, now scores are 109:187
297th round, Strategy1: R, Strategy2: S, Strategy1 is winner, now scores are 110:187
298th round, Strategy1: R, Strategy2: S, Strategy1 is winner, now scores are 111:187
299th round, Strategy1: R, Strategy2: P, Strategy2 is winner, now scores are 111:188
300th round, Strategy1: R, Strategy2: P, Strategy2 is winner, now scores are 111:189

In the end winner is <__main__.Epsilon_Q_Learning_Strategy object at 0x7fb3a81f6610>
.          loser is <__main__.Epsilon_Strategy object at 0x7fb3a81f66d0>
```

Figure 7. Competition Results 1.

4.3. ϵ-Greedy Strategy based on Q-Learning Update Agent v.s. AC Agent

To our surprise, the performance of AC Agent was not as excellent as we expected, it is super sensitive to the given parameters and is suffering from long convergence time of neural networks and thus turns out not to be an ideal algorithm to play RPS with a rival with varying strategy. Remember the metaphor of child and parent in Section 3.5, instead of a child play around in a unchanging garden, now it's more like the child is running ahead in a road that the

parent is unfamiliar with, therefore though the parent are trying to guide his/her child, the environment are rapidly changing, and thus the parent couldn't give best instruction for all the time.

Take ϵ-Greedy Strategy based on Q-Learning Update agent in this experiment for instance, though it might have a certain tendency for several rounds, thanks to ϵ, its policy is changing all over the rounds. The adaptation of AC agent is restricted by the neural networks parameters update speed, as the AC neural network is fully converged, the rival's policy might also be changed already, which results in the "dullness" of AC algorithm. To accelerate the optimization, it sounds good to add its learning rate to higher values. While it is not the case, a NN model with a high learning rate faces the risk of overfitting, with loss never reduced to the global minimum.

```
995th round, Strategy1: P, Strategy2: R, Strategy1 is winner, now scores are 510:485
critic_value [[0.39803913 0.32101429 0.28094658]]
996th round, Strategy1: P, Strategy2: R, Strategy1 is winner, now scores are 511:485
critic_value [[0.44806814 0.28174296 0.27018896]]
critic_value [[0.7516217  0.24014139 0.00823686]]
997th round, Strategy1: R, Strategy2: P, Strategy2 is winner, now scores are 511:486
critic_value [[0.76898336 0.17801525 0.05300139]]
998th round, Strategy1: S, Strategy2: R, Strategy2 is winner, now scores are 511:487
critic_value [[0.5796192 0.1366175 0.2837632]]
999th round, Strategy1: P, Strategy2: R, Strategy1 is winner, now scores are 512:487
critic_value [[0.46946484 0.22583339 0.30470183]]
1000th round, Strategy1: P, Strategy2: R, Strategy1 is winner, now scores are 513:487

In the end winner is <__main__.Epsilon_Q_Learning_Strategy object at 0x7f9e5c8a6390>
.          loser is <Actor_Critic_Strategy.AC_Strategy object at 0x7f9e5c8a6f10>
```

Figure 8. Competition Results 2.

4.4. LSTM based on Deep Q Network Strategy Agent v.s. Human and Sequence input

The LSTM cell model summary is shown in Fig. 9. The results of human v.s. LSTM based on DQN are shown in Fig. 10, we can see that it is not hard to win or tie in the end(with 0.34 win rate for human and LSTM model both). This is very different from the results we expected. In order to find the key factor, we design another experiment changing the input as a fixed "rock", to lower the difficulty for the LSTM model. The results are shown on the Fig. 11, we can see that the LSTM model could adapt to the fixed input and hit a win(LSTM model win rate 0.58 and fixed input win rate 0.22) after 2 epochs with each 100 moves. When we change the sequence to the form "rpsrpsrpsrps…", the model still could generalize and win at last. So we can infer that the deep model needs more time to adapt to the change of environment, if the opponent suddenly changes the policy, LSTM model might need more than 20 time step to detect the change and make reaction.

4.5. Markov Chain Agent v.s. Human

The left of Fig. 12 shows the last 4 steps of a 50 round RPS game with Markov chain model, though we won the last 3

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 50)                12000

 dense (Dense)               (None, 3)                 153

=================================================================
Total params: 12,153
Trainable params: 12,153
Non-trainable params: 0
_________________________________________________________________
None
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_1 (LSTM)               (None, 50)                12000

 dense_1 (Dense)             (None, 3)                 153

=================================================================
Total params: 12,153
Trainable params: 12,153
Non-trainable params: 0
_________________________________________________________________
None
```

Figure 9. Model summary of LSTM cell model.

steps, but still can't win in the last. Additionally, most of those who tried this game told me that they could find the logic and law that the algorithm follows, if given them more chances they could beat the Markov chain model. So I set the game round to 100 and results are as shown in the right of Fig. 12, they still can't win. Because the model has a "random protection" function, it will randomly output for 5 rounds and mislead the human player. When the human player realizes the model has gone back to the pattern, he has already lost for more than 10 rounds. Therefore, more rounds played, lower the winning rate. You can also try it yourself by opening the markov_chain_vs_you.py in the google colab. But I have to admit that the Markov chain model is still using people's strategy and reinforcement learning only provides a logical method (state transition matrix) to give the prediction, so it's still a contest between people, it's not a total artificial intelligence.

```
The 47 round of the game:
please input your choice r/p/s:s
Your rival choice: r
Result : YOU LOST
Your win rate is: 0.3404255319148936
LSTM win rate is: 0.3617021276595745
The 48 round of the game:
please input your choice r/p/s:s
Your rival choice: s
Result: TIE
Your win rate is: 0.3333333333333333
LSTM win rate is: 0.3541666666666667
The 49 round of the game:
please input your choice r/p/s:r
Your rival choice: s
Result : YOU WIN
Your win rate is: 0.3469387755102041
LSTM win rate is: 0.3469387755102041
The 50 round of the game:
please input your choice r/p/s:s
Your rival choice: s
Result: TIE
Your win rate is: 0.34
LSTM win rate is: 0.34
EPISODE  1
 P1 rock rate: 0.42 paper rate: 0.30 scissors rate: 0.28
 P2 rock rate: 0.28 paper rate: 0.28 scissors rate: 0.44
 Avg reward: 0.34 Avg Qmax: 0.08
```

Figure 10. LSTM model v.s. Human results.

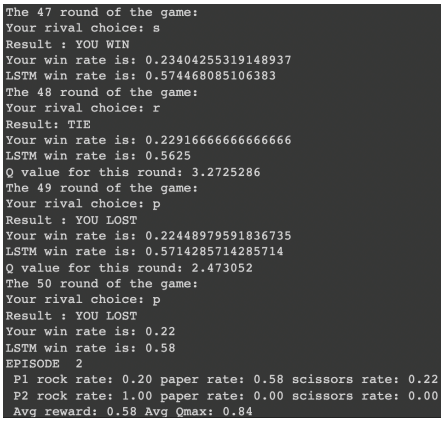

Figure 11. LSTM model v.s. Fixed input "Rock".

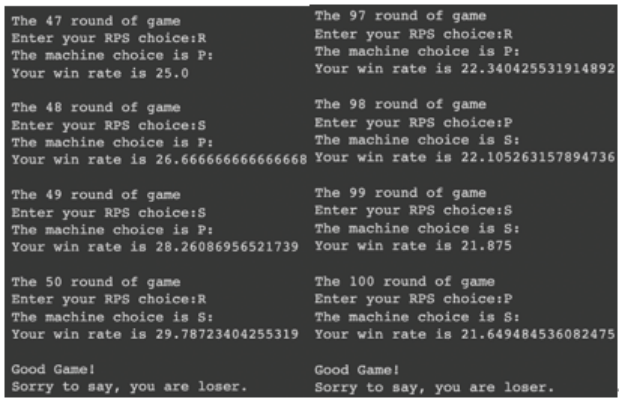

Figure 12. Game round = 50, 100 Markov Chain model.

5. Conclusion

The purpose of this final project was to identify optimal strategies for playing Rock Paper Scissors(RPS) games. Considering the simple environment and state action pairs, at first glance it might seem that algorithms are inapplicable to RPS. Nevertheless, along with deeper research in the field of RPS, we recognized that with concrete state, actions and reward defined by the rules, RPS could serve as a platform for multiple reinforcement learning strategies.

Basically our final project is carried out in the following four steps:

1) Compose straightforward strategies like Markov Chain and epsilon-greedy to construct the model and test its stability by interacting with them on terminals.

2) Implement more profound reinforcement learning algorithms learnt from class such as Q-learning.

3) Set up multi-agents competition environment so that they can play with each other.

4) Build deep reinforcement learning models, including the LSTM model based on DDQN and Actor-Critic strategies which are expected to generate better performance.

Based on the results shown in Section 4 and analysis conveyed, it can be concluded that:

1) The ϵ-Greedy strategy based on Q-Learning performs better than simple ϵ-Greedy Strategy for the Q-matrix enables the former a long term "memory" that keeps updated over all match play rounds. 2) The performance of Actor-Critic strategy is slightly below the expectation because not only the size of the training sample is small compared to a traditional neural network, but also the random variation of opponents strategy e.g. ϵ of ϵ-Greedy strategy based on Q-Learning algorithm. The restriction from settings of hyperparameters and speed of convergence also troubles the performance of the AC agent.

3) The Markov Chain model experiments demonstrate that the environment of the Markov chain could be the platform for multi strategies generated by humans. It could provide a proper and reasonable method to give the prediction from the state transmission matrix. In consequence, this could just be a half reinforcement learning agent, because its strategies are set by humans.

4) The LSTM cell model based on DDQN experiments illustrates the limitation of deep learning models handling the dynamic environment. It requires a fixed environment, like the fixed sequential input, to have a good generalization property. It is not hard to understand, as is known to all, a deep learning network needs tons of training dataset to obtain a better ability to generalize. But our LSTM model is trained during the play, which means it could adapt to some pattern of input but if the input pattern suddenly changed, the LSTM model may need more than 20 time steps to modify the hyperparameters to adjust itself to the new environment. Therefore, if a much bigger model is built and tons of input patterns are used for training I believe that the LSTM model could generalize to the dynamic environment.

Note that the "upgraded" version of the RPS game, Rock, Paper, Scissors, Lizard and Spock(RPSLS) is also explored in our project, while constrained to the limit of pages we decided to upload it to Google Colab, you can find the link in README and play with our RPSLS robot. compared with RPS, RPSLS has a much less chance of draw, widens the performance gap between algorithms in same play rounds, and is worth further exploration.

Reference

(1) Zhang, Hanshu & Moisan, Frederic & Gonzalez, Cleotilde. (2021). Rock-Paper-Scissors Play: Beyond

the Win-Stay/Lose-Change Strategy. Games. 12. 52. 10.3390/g12030052.

(2) Li, C., & Qiu, M. (1970, January 1). Reinforcement learning for cyber-physical systems (2019 edition). Open Library. Retrieved December 21, 2021

(3) Chan, Ka & Lenard, C. & Mills, Terence. (2012). An Introduction to Markov Chains. 10.13140/2.1.1833.8248.

Contribution of Team Members

Hanshan Li: Set up epsilon-greedy, epsilon-greedy based on Q-learning model, Actor-Critic strategies for both RPS game and RPSLS game(Quaternary model)and set up environment for these 3 models to engage in confontation, composing the section 3,4,5 of final report.

Chongzhi Xu: Set up LSTM cell model based on DDQN, establish the Markov Chain model for RPS game(play with human) and set up environment for LSTM and Markov to engage in confrontation, composing the section 3,4,5 of the final report.

Zifan Li: Writing section 1 and 2 of the final report, arranging the final report in latex.