# Shooting King

Team 15
Wang Tzu-Wen, Tsao Li-Yuan, Liao Huang-Ru, Ou Chuan-Ming, and Wang Ling-Sung

***Abstract*—Our goal in this project is to train a smart RL agent to play Rock, Paper, Scissors with us. Also, we want to use the computer vision model to detect the gesture of Rock, Paper, or Scissors in real-time using the laptop's camera. So the computer can know the player's move and learn or count the score without we type in the result. And it's fun that we might be able to build a model detecting what the gesture is in real-time and cheat using that information.**

## I. Introduction

ROCK paper scissors has been a staple to settle playground disagreements or determine who gets to ride in the front seat on a road trip. The game is simple, with a balance of power. There are three options to choose from, each winning or losing to the other two. In a series of truly random games, each player would win, lose, and draw roughly one-third of games. But people are not truly random, which provides an opportunity for AI to understand a player's tendencies and beat them.

### A. Reinforcement Learning

To win more in this game, the most intuitive way is to guess your opponent's policy, once you know your opponent's policy, you can easily beat him by taking the winning action. So, many people make lively discussion and research in order to find the best way to analyze various policies around different people.

There is just a contest on Kaggle with the Rock, Paper, Scissors problem, many people came up with some interesting and clever ways to be the policy of their agent. Here are some cool ways we've seen in Kaggle's discussion. One team collects the result of a great amount of games as their dataset, and they took a random forest classification model as their agent, this model got a silver award in the Kaggle contest. Another team used the LSTM technique and trained on batch to analyze the sequence of their opponent's actions, which got a bronze award of the Kaggle contest. Another strong team used the multi-armed bandit approach, which is an advanced technique of reinforcement learning to select the most promising counter-strategy and use it to beat the opponent, we have heard the concept but not familiar with it, and they got a gold award, which is the best result of a Kaggle contest. Many teams use the concept of 'memory' as part of their agent's policy, we are inspired by them and choose the Markov Decision Process as our core concept, which stores the transition probability matrix so that we can sample our action according to the transition probability matrix.

### B. Gesture Detection

After we have an agent to play rock, paper, scissors with us, we need to build an interface that people can interact with our agent in real time. So, our computer needs to understand our gestures and know who is the winner. To achieve this goal, we adopt the MobileNet model to identify our gestures. Mobile Net is a light weight classification architecture which can classify the gestures in real time. Then, our agent receives the result and updates the policy for next turn.

## II. Methods

TO build a rock, paper, scissor robot, we use an image classification model to classify our gesture, then we use an reinforcement learning model to predict the player's method. Finally, we concatenate two models to build a robot. Below we discuss these two models.

### A. Reinforcement Learning

*1) Markov Chain:* In a reinforcement learning (RL) problem, we use some technique and some algorithm to train a smart agent in order to receive a great amount of reward, in other words, to complete a real task. For our RL agent, the opponent of the player, we take the concept of Markov Decision Process (MDP) into our model, which is a fundamental concept of RL problem. In a MDP environment, it stores a tuple of $(S, A, P, R, \gamma)$, where $S$ is a set of states, $A$ is the set of action that our agent can choose to take, $P$ is the transition probability matrix, $R$ is the accumulated reward and $\gamma$ is the discounted factor. The agent will sample an action according to the state, and the transition probability matrix, and it will be trained for the purpose of getting as much reward as it can.

Here we take the action pair of the player and our agent at the previous step as our state. We build a transition probability matrix like TABLE I shows. There is also a matrix similar to transition probability matrix, but is store number of time a action is observed. We named it $nobs$ here.

TABLE I: Transition Probability Matrix

| State | R | P | S |
|---|---|---|---|
| RR | 0.33 | 0.33 | 0.33 |
| RP | 0.33 | 0.33 | 0.33 |
| RS | 0.33 | 0.33 | 0.33 |
| PR | 0.33 | 0.33 | 0.33 |
| PP | 0.33 | 0.33 | 0.33 |
| PS | 0.33 | 0.33 | 0.33 |
| SR | 0.33 | 0.33 | 0.33 |
| SP | 0.33 | 0.33 | 0.33 |
| SS | 0.33 | 0.33 | 0.33 |

The matrix stores the probability of what is the next action the player will take according to the current state. So our main loop Algorithm 1 agent call $predict$ to sample the action that the player most likely to take at the next time step. And choose the action which can beat the player. After one round, we update the transition probability matrix according to the state of the one previous round and what action the player actually took in this round. We also decay the probability of other state-action pairs by the discounted factor $\gamma$, so our transition probability matrix can be more robust to fast-changing policy of the player.

**Algorithm 1:** Main Loop

```
Initialize MDP Matrix;
while step < MAX-STEP do
    agent.predict();
    Player show his play p;
    agent.update(p);
    Print who win the game;
end
```

The $update$ algorithm show in Algorithm 2. It's input is previous state (previous player's play + previous agent predict) and the current player's play. We first multiply the previous state's $nobs$ matrix by discount factor $\gamma$. Then increase the corresponding $nobs$ value by 1. Finally, we calculate the transition probability using $nobs$.

**Algorithm 2:** update

**input :** $s_{previous}$, $d_{player}$
$M[s_{previous}]$[nobs][:] *= $\gamma$;
$M[s_{previous}]$[nobs][$d_{player}$] += 1;
$total$ = Sum($M[s_{previous}][nobs][:]$);
$M[s_{previous}]$[prob] = M[$s_{previous}$][nobs] / total;

*2) Number of look back step:* During the training process, we encountered some problems while using the basic Markov Decision Process with the state which only has the information about the action pair of the player and our agent just one previous step. To validate the performance of our agent, we design some policy of the player, for example, a player only chooses 'rock' as his action at every step, or some a little bit complex but still fixed policy like having some repeated pattern. For those simple policies like only taking 'rock' as his action, our agent can learn the policy of the player soon and easily beat the player, but if the repeat frequency of the pattern of the player's action is much longer, it is hard to have good performance for our agent.

Therefore, we started to think about the reason for the problem. We concluded that if there really exists some implicit policy of the player, if we take more previous steps of the player's action into consideration, maybe we can get more useful information for it. So we tried to look back more step of the action pair of the player and our agent as the state, hoping to build up a more robust transition probability matrix in order to make a more precise prediction. Here we define the number of step we look back as $level$. We found that if $level$ increases just from one step to two steps, it makes a huge improvement, which can easily beat some more complicated pattern that the one-step-look-backward agent can not handle. Finally, we choose the multi-step-look-backward agent to solve the Rock, Paper, Scissors problem, the result of our experiment will be shown later.

### *B. Gesture Detection*

*1) Dataset Preparation:* Our dataset consists of the gestures of the game: Rock, Paper, Scissors. First, we prepare 300 images of training data for each gesture. However, despite the accuracy through the training is quite great, the model seems to overfit the training dataset. To solve the problem, we adjust the amount of the training data to 500 images and prepare 100 images of validation data respectively. Because we use the mobilenet as the pretrained model, the size of the image needs to be resized in 224 * 224 in order to correspond to the input size. We use the webcam to collect the dataset. To produce the dataset for the gestures, we show every gesture in different kinds of angles and positions dynamically in front of the webcam, then the camera will continuously capture our gestures, transferring every frame to the image.

*2) Dataset Preprocessing:* First, we train the model without augmentation. However, the model seems to overfit the training dataset. To solve this problem, we do augmentation in our dataset to prevent our model from learning the specific pattern. The augmentation includes random flip_left_or_right to increase the diversities of the angles and positions, and random pad 10 pixels in both width and height, then random crop to 224 * 224 for the diversities of background and robustness outcome of incomplete images.

*3) Model architecture:* Our model architecture is defined as Table I. We use the pretrained MobeilNetV1, and add a global average pooling and a fully connected layer in the end. Since our task is rather simple with only three classes: rock, paper, scissors, we add just one layer to prevent over-fitting. Mobile networks are the most commonly used techniques for modern deep learning models to perform a variety of tasks on embedded systems. MobileNet architecture has 28 layers in total where each layer except the last fully connected layer is followed by a batch normalization layer and ReLU activation layer. It is based on depth wise separable convolutions which is a form of factorized convolutions which factorize a standard convolution into a depthwise convolution and a 1×1 convolution called a pointwise convolution. We use it to perform real-time detection since it is efficient and fast to predict. The network architecture is shown in TABLE II.

*4) Game Interface:* We develop a game interface for the demonstration of our real-time model system. We use the webcam to capture frames in a short period. The frame will be sent into our model, then the model will produce the final result immediately. The whole progress is described as below. First, the user can show some gestures to see the model prediction. When the game starts, the user needs to make his/her final decision before counter counting down from three to one. We will send the last frames into our model, and show

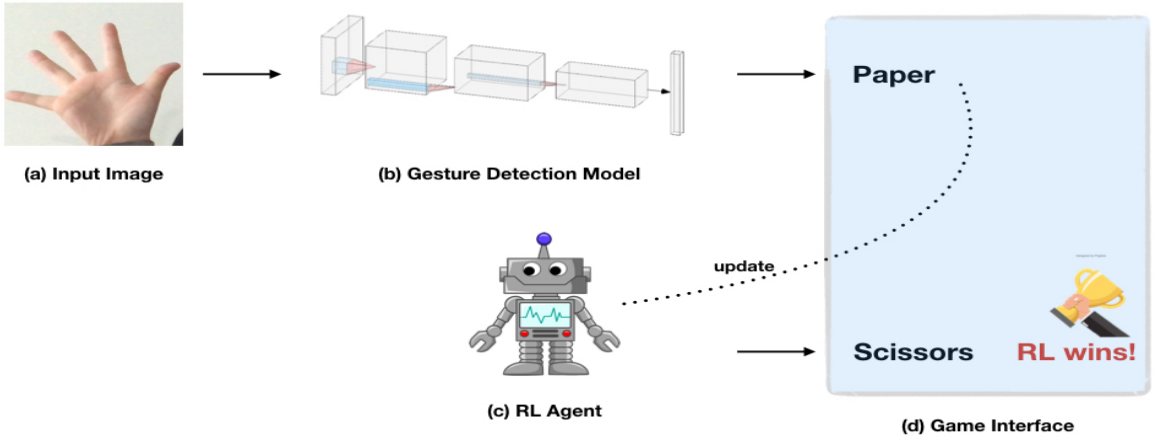



Fig. 1: Shooting King Structure

the classification result on the screen. At the same time, the computer will also make its decision. Then the game result will be calculated and shown on the screen. The user can press the specific key to reset the game. Besides RL agent, we also develop 2 other different modes, which is random mode and cheat mode. In random mode, the computer will take action randomly. In cheat mode, the computer will take action depends on the user's action, so it will definitely win the game. The game interface structure includes Ready, Prediction, Show_Result modules. In Ready, the frames will be kept sent to our model, and showing the detection result. In Prediction, we use tensorflow package to load our model saved in h5 file, and make the prediction. In Show_Result, we draw some text and decoration to display the result. We use the OpenCV package to implement our game interface, which includes combining webcam to capture frames per time, showing text on screen, and key press detection.

TABLE II: NetWork Architecture

| Layers | Output Size |
|---|---|
| Input | 224*224*3 |
| MobileNet | 7*7*1024 |
| Global average pooling | 1024 |
| Dropout | 1024 |
| Class | 3 |

*5) Shooting King Structure:* Our Shooting King Structure includes two main parts: RL and Gesture Detection. First, the camera catches the frames and sends it into the Gesture Detection model, the model predicts its detection result. Then, the RL agent will take its action. Finally, the Game engine will calculate the final result and show it on the screen. Figure 1 shows the whole structure of our Shooting King.

## III. Results

HERE we show our experiment detail and result of both reinforcement learning part and Gesture classification part.

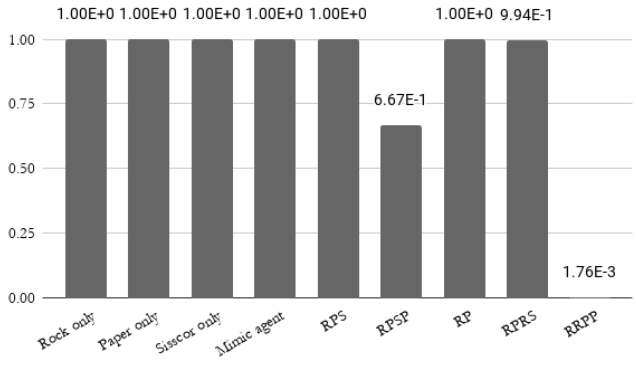
### A. Reinforcement Learning

We have done some experiment to verify our agent's performance. Firstly, we compare different level's performance. Secondly, we will show that as level increase, the time to converge also increase. Finally, we let our agent to play with human.

*1) Naive Agent:* The simplest agent only take one previous state as input (i.e. level=1). And we have design some opponent with specify pattern to play with our agent. Below is the patterns we design.

- Play one type only: play one of R, P, S only.
- Mimic agent: Mimic what agent play last time.
- RPS: Play rock, paper, scissor in turn.
- z-shape (RPSP): Play rock, paper, scissor, paper in turn.
- RP: Play rock, paper in turn.
- RPRS: Play rock, paper, rock, scissor in turn.
- RRPP: Play rock, rock, paper, paper in turn.

Fig. 2: Level 1 Agent vs Different Opponent

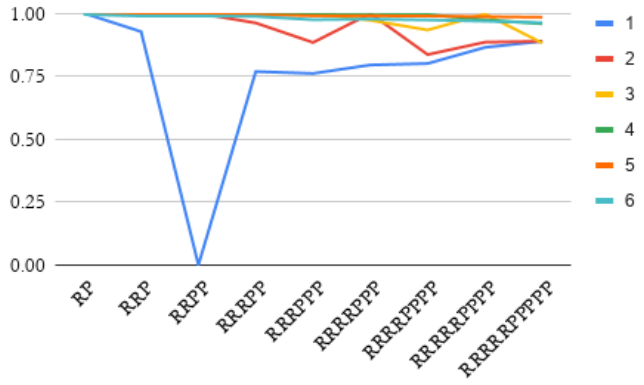



In Figure 2 we can see that this simple agent can beat most pattern with 100% winning rate. However, playing with pattern "RPSP", our agent only get 66% of winning rate. And our agent is beat by the pattern "RRPP" completely. We notice these phenomenon is related to the opponent's pattern cycle. If the cycle is bigger than 2 and there are duplicate play in the

pattern, than our agent can't reach 100% winning rate. That is because this agent only take previous state as input. For example, if the opponent's pattern is "RRPP", and previous state is "R". Then the next state can be either "R" or "P". So it's imposible to give accurate answer. As the result, we increase the level, that is, take more previous steps as input. In this way, our agent can learn pattern with longer cycle.

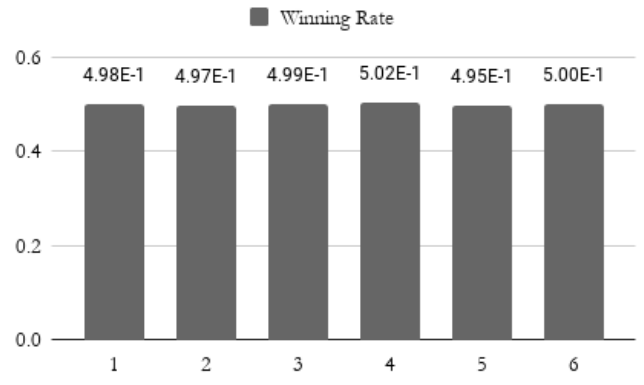
*2) Different Level:* Here we use agent with larger level. And we test them with different pattern. The pattern has different cycle as well. Figure 3 shows the result. As we can see in the figure, all agent can handle pattern with cycle 2. But as pattern cycle increase, lower level agent's winning rate start to drop.

Fig. 3: Different Level Agent vs Different Opponent




*3) Random Opponent:* As we can see in Figure 4. All agent can only reach around 50% of learning rate. Which it's a reasonable result.

Fig. 4: Different Level Agent vs Random Opponent




*4) Agent vs Human Performance:* Our final goal in this experiment is to build agent to beat human. So we've done an experiment test the agent's winning rate when playing with real person. We use level 3 agent here. And playing with person for 40 rounds. Our agent got winning rate around 50%.

## B. Gesture Detection

Our gesture detection model is trained on 1500 training images, and evaluated on 300 validation images. We evaluate both accuracy and loss. The image is resized to 224 * 224, with random flip, crop and pad augmentation. We initialize the weights with the pretrained weights on ImageNet. We use Adam optimizer with mini-batch size of 32. The learning rate is set to be 0.001 without decay. The models are trained for up to 100 iterations, but we observe over-fitting for more epochs. Besides the original version, we also make some changes to our net for comparisons.

Figure 5, Figure 6 is our training result of loss and accuracy on both training and validation data. We can find that the loss on validation increase dramatically after the second epoch, so does accuracy. So we decide to train our model for 2 epochs.

Fig. 5: MobileNet accuracy curve in training dataset and validation dataset

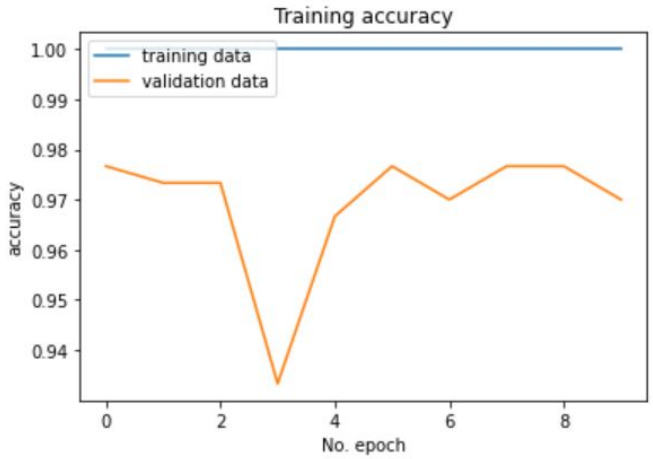



Fig. 6: MobileNet loss curve in training dataset and validation dataset


We have done some experiments to verify our model performance. First, we show the results for our model with

different Dropout rates. In Table III, we see that using 0.5 is better than 0.2. The result corresponds to our expectation because most of the papers we survey use the Dropout rate 0.5 .

TABLE III: Dropout 0.5 vs Dropout 0.2

| | F1 score | Accuracy |
|---|---|---|
| Dropout 0.5 | 0.99 | 99% |
| Dropout 0.2 | 0.9867 | 98.66% |

Next, we show the comparison of whether fine tuned on our dataset. In Table IV, we see that the performance of fine tuned is a lot better than the not fine tuned. This is because after fine tuned the model will become more suitable to our dataset.

TABLE IV: Fine tune vs Without Fine tune

| | F1 score | Accuracy |
|---|---|---|
| Fine tune | 0.99 | 99% |
| Without Fine tune | 0.6778 | 72.67% |

Then, we show the comparison of adding FC or not. The FC we use is the relu activation function with different units. In Table V, we see that the performance of adding the FC is not as good as our model (not adding FC). We believe that the reason is our classification is quite simple, so adding the FC is just the opposite to what we wished, making the model overfit and causing the bad performance.

TABLE V: Original vs FC unit 10 vs FC unit 100

| | F1 score | Accuracy |
|---|---|---|
| Original | 0.99 | 99% |
| FC unit 10 | 0.9474 | 94.67% |
| FC unit 100 | 0.9824 | 98.67% |

Last, we show the comparison of using different real-time pretrained models. In Table VI, we see that Vgg16 has the worst accuracy and the lowest FPS, and although Resnet has good accuracy and FPS, MobileNet is still better than Resnet on either side. Hence, MobileNet is the best choice for our project.

TABLE VI: MobileNet vs Vgg16 vs Resnet

| | F1 score | Accuracy | Real Time |
|---|---|---|---|
| MobileNet | 0.99 | 99% | 8FPS |
| Vgg16 | 0.9666 | 96.67% | 1FPS |
| Resnet | 0.98 | 98% | 7FPS |

## IV. Discussion

### A. Reinforcement Learning

The naive Markov model can't learn all pattern in our experiment. But larger level agent can beat longer cycle pattern. Also, when playing with human, our agent can only reach winning rate slightly over 50%. We believe our agent can't beat human since we only train agent for 40 round. Without large training data, agent can't learn human's pattern completely.

### B. Gesture Detection

Since our training model has high accuracy, we expect that the model can classify all the gestures from the real-time stream. However, during the testing time and the demo time, we found that the background of the frame plays a significant role in the accuracy of the classification. If the background color is not monochromatic or white, or maybe the background consists of multiple objects like windows, door, or cabinet instead of the walls, like Fig 7, chances are the model will be confused with the rock and scissors gestures, which leads to the poor accuracy of the classification.

Fig. 7: Classify Rock into Paper

Furthermore, the brightness of the background is also a critical issue. If the source of light is not sufficient, the gestures will become darker and maybe blend with the background, like Fig 8, which also causes the bad performance. Hence, we speculate that the diversity of our images in the dataset isn't enough. And to be a 'good' dataset is not only to increase the differences of the target object that the model is going to classify, but also the environment of the pictures.

Fig. 8: Classify Scissors into Paper

Besides, we do the augmentation in our dataset through the data preprocessing, so we expect that our model can classify all the gestures in a simple background with arbitrary angles and positions. However, we surprisingly found that if using the left hand to show the gestures, like Fig 9, our model can't classify correctly. Hence, we come to understand that augmentation might not be effective if the dataset itself isn't diverse in the beginning.

Fig. 9: Classify Scissors into Rock

Another problem we encounter is that although the real-time stream is 30 fps, there are possibilities to capture the blurred pictures while the hand is moving, like Fig 10. A trivial idea to solve the problem is increasing the fps in order to decrease the possibilities of the blurred pictures, but the fps depends on the hardware performance, it won't be a general solution for all the users who want to run our project on their own computers. So we came up with another idea: adding the blur function to the data preprocess, making the model also learn the blur image of the gestures. And more importantly, this idea is a general solution because of the independence of the hardware performance.

Fig. 10: Classify Scissors into Rock

## V. Conclusion

We build a rock, paper, scissors robot using image classification model and reinforcement learning agent. For the image classification part, we use mobile net to achieve real time classification. We also tuned some parameter in mobile net and add our own fully connected layer to achieve 99% accuracy on validation set. As for the reinforcement learning part, we use a simple Markov Chain agent to predict human's strategy. And the agent can reach slightly over 50% of winning rate when playing with human. We also develop a game interface with three mode: RL mode, Cheat mode, Random mode. In RL mode, you will play against the agent. In Cheat mode, the computer will take action based on the player's final action, so the player will definitely lose every game. In Random mode, the one who has the better luck wins! We hope our combination of Computer Vision and Reinforcement Learning can spice up the rock, paper, scissors game with more challenging opponent and efficient detection model, which creates a whole new way to play this game.

## VI. Author Contribution Statements

- W.T.W (20%): Implement Markov Chain agent. Reinforcement learning's result and discussion report. LaTeX layout adjustment.
- T.L.Y (20%): Implement the code of Markov Chain experiment, responsible for reinforcement learning's introduction and method report.
- L.H.R (20%): Game Interface, gesture detection's method, result report.
- O.C.M (20%): Train MobileNet model, collect data, gesture detection's result report.
- W.L.S (20%): collect data, gesture detection's method, result and discussion report.

## References

[1] Piotr Gabrys. How to win over 70% matches in rock paper scissors. https://towardsdatascience.com/how-to-win-over-70-matches-in-rock-paper-scissors-3e17e67e0dab, 2020.

[2] Rock, paper, scissors shoot! https://www.kaggle.com/c/rock-paper-scissors/leaderboard.

[3] Andrew G.Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. https://arxiv.org/pdf/1704.04861.pdf, 2017.