Learning to Win Rock-Paper-Scissors

Zhi Bie[*†], **Yang Song**[*‡], **Tong Yang**[*†]
[†] Dept. of Electrical Engineering, Stanford University, CA, 94305
[‡] Dept. of Computer Science, Stanford University, CA, 94305
{zhib, songyang, tongy}@stanford.edu

Abstract

The Nash equilibrium of Rock-paper-scissors is random play. However, this strategy may not be optimal in real cases especially with multiple runs, since it is generally very hard for a person to act randomly, and it is attractive to exploit this non-randomness to gain advantages over the opponent. Our goal is to use machine learning methods to analyze the hidden patterns of the opponent and act accordingly in order to increase the probability of winning. To this end, we tried sequence modeling approaches, *e.g.*, HMM and RNN, behavior cloning methods, *e.g.*, logistic regression and SVM, and reinforcement learning algorithms. We compared and analyzed the pros and cons of different algorithms extensively in the experiments and proposed a bandit method to combine them dynamically. We also implemented a nice dynamic web page to demonstrate our algorithms intuitively.

1 Introduction

Rock-paper-scissors (RPS) is a popular game among children, in which players simultaneously form one of three shapes with their hands. There are three outcomes of the game. A player who has decided to choose rock will win if the opponent shows scissors but will lose to a player who has thrown paper. Similarly, a play of scissors will win a play of paper. The game comes to a tie if both players show the same gesture.

It is obvious to see that the Nash equilibrium of the game is to choose gestures randomly, but is it optimal in practice? In real cases, RPS is usually played multiple runs and the winner is the one who has won a majority of plays. Moreover, people will be psychologically affected by previous wins or loses. Note that the random strategy ensures the player will not lose with a statistically significant margin, but can never guarantee the player any advantage over the opponent either. We suggest, inspired by our own experience of playing this game, that it is generally very hard for people to play truly in a random fashion and there are some hidden patterns behind those plays. For example, [1] has discussed some common psychology phenomena: 1) winners will tend to stick to the winning action; 2) losers tend to change their strategy and move to the next action in R-P-S sequence; 3) stats show that the most common throw is rock (35%), scissors (35%) and then paper (29.6%). There are also books, *e.g.*, [2], that teach you the best strategy based on those findings.

Therefore, the problem of winning rock-paper-scissors has aroused broad interest in the computer science community. A nice example is [3], where the algorithm records the last 4 plays of both sides and then searches over 200,000 rounds of previous experience to compute probabilities of the player's next move. [4] uses simple classifiers like binary SVM and logistic regression to do sequence prediction based on the author's playing history. [5] uses Gaussian Mixture Models (GMM) to predict the next play based on three previous plays. [6] exploits a neural network structure called WiSARD to model the move of the opponent, and also discusses some tricks like bluffing. There are

[*] Authors are listed alphabetically.

also international contests of designing highly effective RPS players, such as [7] and [8]. There are even enthusiasts who have generalized the RPS game to the case of 25 different gestures with 300 outcomes (Fig. 1).

In this project, we propose to use machine learning to capture and exploit the hidden patterns of the opponent so as to increase the chance of winning. We explore approaches of three categories: 1) sequence modeling, which takes the sequences as training data and tries to predict the future act based on historical information. Exemplar algorithms include hidden Markov models (HMM) and recurrent neural networks (RNN); 2) behavior cloning, which takes a fixed number of acts before the current one as features and turn the prediction problem to classification. Typical methods include logistic regression (LR) and support vector machines (SVM); 3) reinforcement learning (RL), which tries to search for the best act based on current state summary and observation of the opponent's policy. We found that sequence modeling / behavior cloning methods and reinforcement learning methods are more or less complementary to each other, so we propose to additionally use reinforcement comparison (RC) [10], a multi-armed bandit algorithm to combine two typical algorithms of different categories.

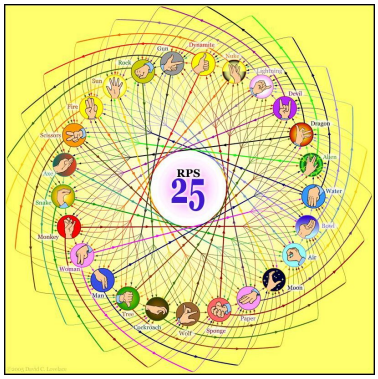

Figure 1: RPS-25 [9]

2 Settings

Input-output There are two agents in the rock-paper-scissors game: the player and the program. Let $\{x_1, x_2, \cdots, x_n\}$ be the gestures given by the player, where $x_i \in \{$'R', 'P', 'S'$\}$, and $\{y_1, y_2, \cdots, y_n\}$ denote the gestures of the program in a similar way. Note that we abbreviate 'R' for "Rock", 'P' for "Paper" and 'S' for "Scissors'. Suppose there are T rounds of RPS in one game, the program needs to give output y_{t+1} based on the history $h_t = \{(x_1, y_1), \cdots, (x_t, y_t)\}$ so as to maximize the total times of winning. We give an example in Tab. 1.

Table 1: An example of input-output

Agent							Winning Times	Winner
Player	R	R	P	S	P	R	1	
Agent	R	P	P	R	R	P	3	✓

Evaluation We use the following ways to measure the performance of our algorithm:

- Submit the program to `RPSContest.com` [7] to see how it ranks on the leaderboard. However, the website has requirements of running time and restrictions of python packages, so it cannot assess every algorithm.
- Collect a large amount of data of human plays and build a program to imitate humans. This can be used to evaluate the average performance of the algorithm against a human opponent.
- Take the 30 highest ranked programs from `RPSContest.com` leaderboard, put them together with our algorithms and run a local tournament.

When running against algorithms in the online tournaments of `RPSContest.com`, it is guaranteed that there are some "idiot" programs with clear repeated pattern. Therefore it is never optimal to choose actions randomly. Also, the top 30 programs in our local tournaments are not random, so it is not the best policy to play randomly either.

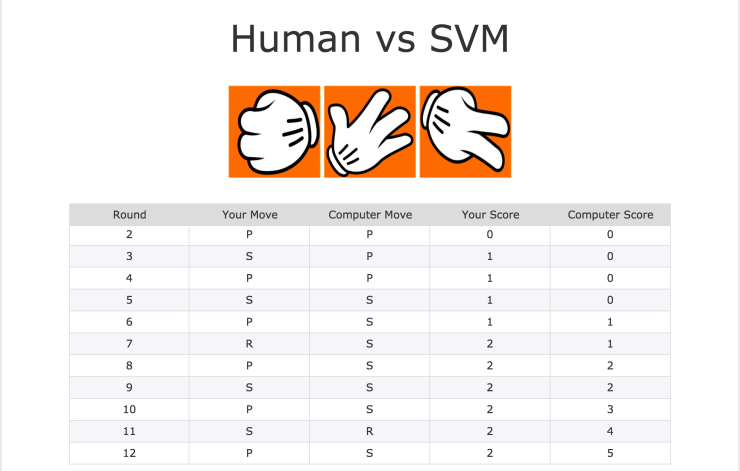

Figure 2: Web page of playing with SVM

Datasets There are 2 kinds of datasets for our machine learning algorithms.

The first one is purely used for evaluating the performance of different algorithms, as indicated above. We have obtained all training data of the New York Times RPS Applet [3] from Prof. Shawn Bayern at Florida State University. The dataset consists of 200,000 rounds of previous playing history of the applet. A sample of it is provided in Tab. 2, where the first and second column record the history of 4 consecutive plays of the player and the computer, and the last three columns record how many times the players have chosen the corresponding gesture.

Table 2: A sample from the NYT RPS Applet dataset

Player	Computer	R	P	S
RRPR	PPPS	46	31	19
RRPR	PPPP	18	36	24
RRPR	PPPR	12	31	14

Although this dataset is extremely useful for playing with humans, it is more important to learn in an online fashion to combat adversarial programs. Note that this kind of data (playing experience with a specific program) can only be gathered during running and are the only source for training our machine learning algorithms.

Infrastructure For local tournaments, we adopt a script from `RPSContest.com` called `rpsrunner.py`, which is able to run one vs one matches of programs. All algorithms are developed in Python 2.7 and possibly involve packages such as `Numpy` and `Scikit-learn`.

We developed a web page to enable humans to play with our algorithms. It was written in JavaScript and HTML/CSS as the front-end, and Python with Flask framework [11] as the back-end. By running `demo.py`, it will set up a server at IP address `127.0.0.1` with port number `5000`. Changing the URL can let us play with different algorithms. For example, `http://127.0.0.1:5000/SVM` leads to an interface of Fig. 2.

For detailed usage of the codes, please refer to `README.md` in the attachment.

3 Approaches

3.1 Challenges

Rock-paper-scissors is an instance of the well-studied zero-sum non-cooperative games and the optimal strategy is to play completely randomly. However, we are surprised by how complicated it can be if real cases are considered. While generating random values seem to be easy for programs, it is generally hard for people to do so. In addition, people are susceptible to psychological affects. Moreover, when considering a tournament where some of the programs (but we do not know exactly which) are guaranteed to be weak, it is never optimal to act randomly. Therefore it has become an interesting task to gain advantage of the opponent, by modeling the patterns and strategies of the adversaries.

The first challenge here is the existence of opponent. Unlike typical decision under uncertainty problems where the environment is independent of the agent, the opponent interacts with the agent closely and will change maliciously according to the behavior of the agent. The states and strategies of the opponent are both hidden from the agent.

The second challenge is randomness. We need to gather useful information from an extremely small number of data, basically those from the immediate history, under very noisy settings.

An interesting fact is the Sicilian reasoning dilemma. The history is obviously visible to both sides, so theoretically the opponent can repeat any reasoning the agent can do about the opponent's strategy, and counteract accordingly. However, since the agent also knows the opponent can reason about how the agent reasons about it, the agent can counteract. This seems to repeat infinitely. In the case of RPS, the depth of Sicilian reasoning is bounded above by 3, since another depth of Sicilian reasoning will lead to the original decision.

3.2 Baseline and oracle

The performance of an agent in this game depends on to what degree it can infer the strategy of the opponent. We take it as the baseline if the agent cannot infer the strategy at all. In this case, the agent should act according to the minimax rule, *i.e.*, assuming the total rationality of the opponent and prepare for the worst. This leads to the Nash equilibrium solution, namely, playing completely randomly. The baseline has a win rate of 47.83% on `RPSContest.com` and is ranked around the middle in the leaderboard. The oracle should be the one who knows the strategy of the opponent exactly, in which case it can counteract the opponent in each step and lead to a complete victory.

Our algorithm should behave between the baseline and oracle. The performance of it measures how strong its inference ability is.

3.3 Sequence modeling algorithms

There are two common sequence modeling algorithms: HMM and RNN. The first one has a standard state-space filtering graphical model and infers the hidden states via forward-backward algorithm. All parameters can be learned by Expectation-Maximization (EM). The latter is a neural network augmented with recurrent connections. It infers the future states by straightforward feedforward propagation and learns the parameters via backpropagation through time (BPTT). We apply both to the RPS game.

3.3.1 Hidden Markov model

Let the length of HMM be N. In the model, the variables are $\{H_t\}|_{t=1}^N$ and $\{E_t\}|_{t=1}^N$, where H_t denotes the hidden variables and E_t denotes the evidence variables. In the game, we take the last N pairs of consecutive moves as the evidence, *i.e.*, $\{E_1, E_2, \cdots, E_N\}$, and update the model parameters accordingly. An example with $N = 5$ is shown in Fig. 3.

We use H_t to represent hidden patterns of the players. Since the pattern can be influenced by various factors, meanings of the hidden state values are not explicitly specified, which shows flexibility of the model. We assume that there are in total K factors that accounts for the patterns of both players, *i.e.*, $H_t \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. The number of evidence states E_t is $3^2 = 9$, representing the number of possible combinations of opponent and player's moves.

After defining the model, we use Expectation Maximization (EM) algorithm to learn start, transition and emission probabilities and predict the next move by forward-backward.

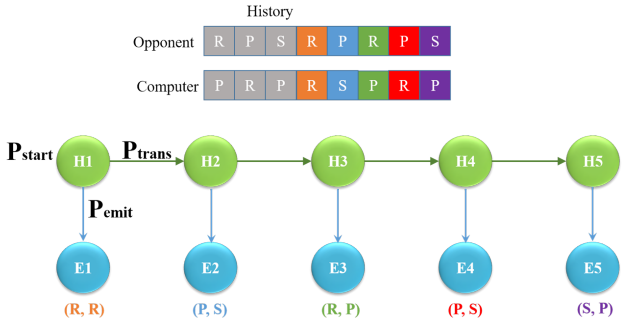

Figure 3: Illustration of HMM.

3.3.2 Recurrent neural networks

In the model of RNN, each move is encoded as a one-hot vector: 'R' $\rightarrow [1,0,0]$, 'P' $\rightarrow [0,1,0]$, 'S' $\rightarrow [0,0,1]$. For each time step t, input x_t represents a pair of move (highlighted in purple in Fig. 4), which concatenates the two vectors and has dimension $1 \times n_{\text{in}}$, where we let $n_{\text{in}} = 6$. RNN maintains a hidden state h_t with dimension $1 \times n_{\text{hidden}}$, through which the memory of the network is propagated and stored. The output o_t (for instance, P in green at time t in Fig. 4) is the prediction of the opponent's next gesture with dimension $1 \times n_{\text{out}}$, where we let $n_{\text{out}} = 3$.

The relationship of input, hidden state and output is as follows,

$$h_t = \tanh(x_t W_{\text{in}} + h_{t-1} W + b_h),$$
$$o_t = \text{softmax}(h_t W_{\text{out}} + b_y),$$

where $W_{\text{in}}(n_{\text{in}} \times n_{\text{hidden}}), W(n_{\text{hidden}} \times n_{\text{hidden}}), W_{\text{out}}(n_{\text{hidden}} \times n_{\text{out}})$ are the weight matrices, and b_h, b_y are the biases. The activation function is $\tanh(x)$. By taking softmax of the output, o_t predicts the probabilities of the opponent's next gestures.

The loss function at time step t is defined as cross-entropy loss:

$$L_t = -y_t \log \hat{y}_t,$$

where y_t is the correct output at time step t, i.e. the input of opponent player at $t+1$, and $\hat{y}_t = o_t$. The parameters in the RNN model is updated using BPTT, and the computer's next move is generated based on the model.

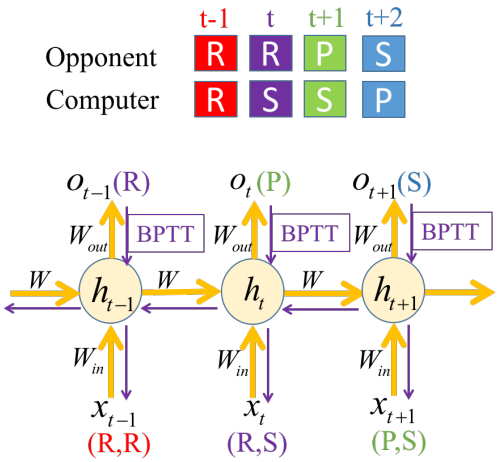

Figure 4: Illustration of RNN.

3.4 Behavior cloning methods

We borrow the term "behavior cloning" from imitation learning literature to describe a group of algorithms that turn sequence modeling to classification problems. After extracting suitable features, every classification algorithm can be used for predicting the future sequence. Closely following [4], we discuss the use of logistic regression (LR) and support vector machines (SVM).

For both algorithms, the features are extracted from the previous N pairs of the opponent's playing history using one-hot representation. For example, if $N = 2$ and the history at time t is $h_1, h_2, \cdots, h_{t-2} =$ 'R', $h_{t-1} =$ 'P', then the feature for predicting h_t is $[1, 0, 0, 0, 1, 0]$. When $t < N$, we use the representation $[0, 0, 0]$ for missing values.

Both algorithms are implemented using `Scikit-learn`. For LR, we use a one-vs-all framework for multi-class prediction, and for SVM, we use one-vs-one. In addition, we use the Gaussian RBF kernel for SVM. Note that although we basically follow all settings in [4], our vital difference is that we do not train on outside datasets. Instead, we only train the model on the historical data in each game play.

3.5 Reinforcement Learning

To apply reinforcement learning, we formulate the game as a Markov Decision Process (MDP). In our case, action $\mathcal{A} = \{$'R', 'P', 'S'$\}$, and reward $\mathcal{R} = \{1, 0, -1\}$. Since we assign reward 1 if the agent wins one round, -1 if loses and 0 if ties, the reward function R is obtained as soon as the states $\mathcal{S}$ have been specified. Here we have tried three different state representations:

1. $S_t = \{(x_{t-N+1}, y_{t-N+1}), (x_{t-N+2}, y_{t-N+2}), \cdots, (x_t, y_t)\}$, where N is the fixed window size. Such states contain a fixed number of moves of both our agent and the opponent. We assume that the opponent's pattern does not depend on moves of time $t - N$ and before. In our experiments we used $N = 4$, and the corresponding algorithm is named RL_table.

2. $S_t = \{(x_{t-N+1}, y_{t-N+1}), (x_{t-N+2}, y_{t-N+2}), \cdots, (x_t, y_t)\}$, but this time we use function approximation to calculate $Q(s, a)$. Specifically, we adopt a map $f = \{$'R' $\to [1, 0, 0]$, 'P' $\to [0, 1, 0]$, 'S' $\to [0, 0, 1]\}$ and let $Q(s, a) = \sum_{i=t-N+1}^{t} w_{ai1} f(x_i) + w_{ai2} f(y_i)$. Note that here we use different weights w_a for different actions a. Using function approximation does not reduce the number of states, but it promotes generalization between states and effectively reduces the amount of data needed for training. In contrast, the previous table-based method needs at least one training instance to train each different state configuration. Therefore, we can use function approximation to train much larger N compared to RL_table. In our experiments, we used $N = 10$ and the resulting algorithm is named RL_approx.

3. $S_t = \{(x_{t_1}, y_{t_1}), (x_{t_2}, y_{t_2}), (x_{t_3}, y_{t_3})\}$. Here we are using a smarter design of states. At time t, t_1 is obtained by searching the most recent history for pattern $\{x_{t-i+1}, x_{t-i+2}, \cdots, x_t\}$ where $i \leq 5$. For example, in Fig. 5 the opponent's last 5 moves are 'PSRPS'. We search for the most recent occurrence of same pattern and take the pair of moves after it, ('R', 'S'), which is highlighted in purple. Similarly, we search the pattern $\{y_{t-i+1}, y_{t-i+2}, \cdots, y_t\}$, $i \leq 5$ to get t_2 and search $\{(x_{t-i+1}, y_{t-i+1}), \cdots, (x_t, y_t)\}$, $i \leq 5$ to get t_3. This reflects our empirical observation that the opponent tends to repeat history. This algorithm is called RL_features.

Based on the model, we use Q-learning to search for the best policy and take the next move.

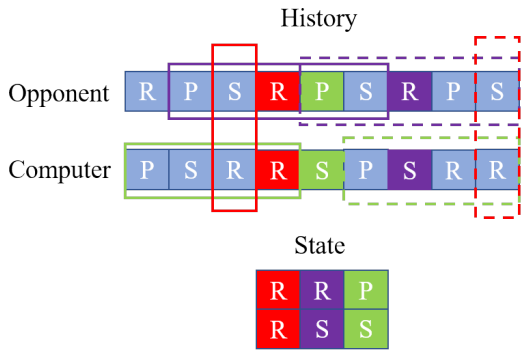

Figure 5: Illustration of the states of RL_features.

3.6 Reinforcement comparison

As would be evidenced in the Experiments section, algorithms of sequential modeling or behavior cloning are more or less complementary to those of reinforcement learning. We desire to combine the different power of all categories while trying to avoid potential pitfalls.

One of the possible ways to do this is by using multi-armed bandit models. Here we view the arms as different algorithms. By pulling different arms, *i.e.*, applying different algorithms to obtain a prediction of the future, we get a reward depending on whether the algorithm predicts correctly or not. This serves to be a nice approximation to what we are for: finding which algorithm is performing better in each step and which is worse, in order to enjoy the pros and avoid the cons.

We propose to use reinforcement comparison [12], a nice solver for multi-armed bandit problems especially when the number of arms is small. More formally, we maintain a preference value $\pi_i(t)$ for each algorithm i. At each turn, we compute the probability $p_i(t) = \frac{e^{\pi_i(t)}}{\sum_{j=1}^{k} e^{\pi_j(t)}}$. If algorithm $j(t)$ is applied at time t and the reward is $r(t)$, the preferences $\pi_{j(t)}$ is updated as $\pi_{j(t)}(t+1) = \pi_{j(t)}(t) + \beta(r(t) - \bar{r}(t))$ and the mean of rewards is updated as $\bar{r}(t+1) = (1-\alpha)\bar{r}(t) + \alpha r(t)$.

In the experiments, we used the algorithm SVM and RL_features as our arms. We fixed the parameters as the recommended defaults in [10]. The name of the resulting algorithm is abbreviated to RC.

4 Experiments and analyses

RPSContest leaderboard: Since the website does not support common machine learning libraries such as `numpy` and `scikit-learn`, and the server has a low-performance CPU with very strict time limits of each match, we could only submit reinforcement learning algorithms to the leaderboard. The results are briefly summarized in Tab. 3.

Table 3: Results on RPSContest website

Program	Matches played	Win rate
Random	631	48.18%
RL_table	205	70.24%
RL_approx	204	52.45%
RL_features	206	76.21%

In Tab. 3, Random is the baseline. Obviously, all reinforcement learning algorithms outperform the baseline, indicating that they can successfully exploit the hidden patterns and strategies of the opponent. The reason why RL_approx did not work is clear: a linear combination of one-hot representations of history is a poor representation of $Q(s, a)$. Since RL_features uses more concise

Rank	Program name	Author	Matches played	Win rate	Rating
1	RVL V3	Thatguy	598	80.6%	8361
2	rps30	teleds	208	79.33%	8270
3	zai_switch_markov1_ml	zdg	566	82.69%	8164
4	metameta v1	Kyle Miller	641	79.56%	8017
5	RL_features	Yang-12	77	81.82%	8013
6	bayes13	pyfex	4292	80.92%	7992
7	fitbot	Flight Odyssey	436	81.65%	7987
8	scattershot	rspk	505	77.82%	7981
9	dllu1_defensive	dllu	614	78.99%	7912
10	IO2_fightinguuu	sdfsdf	493	77.69%	7912

Figure 6: Screenshot of our best preformance algorithm on RPSContest learderboard.

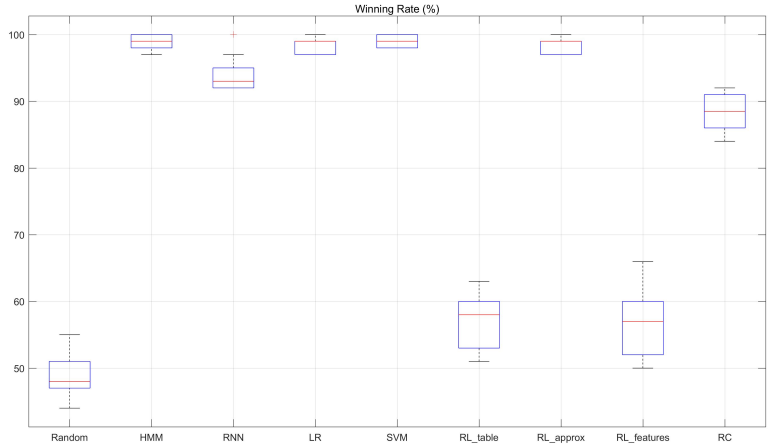

Figure 7: Average human evaluation results

and related features to represent the states, it outperforms RL_tables and RL_approx. Actually RL_features performs so well that it was once ranked 5th/2289 on the website (as in Fig. 6)

A take-home message of this experiment is that the performance of reinforcement learning algorithms on this task depends closely on the qualities of state representations. Intuitively, RL_approx has the worst representation due to linearization, although it might incorporate and generalize to longer time scales via fewer parameters. The state representation of RL_table is limited to the window size N. In contrast, the state representation of RL_features is carefully hand-crafted and reflects our observation of repeated patterns. As a result, the performances of RL_approx, RL_table and RL_features increase.

It might be useful to point out that the ranking of `RPScontest` depends on some score different from the winning rate. That score puts more weights on recent victories and is sometimes not stable since not all algorithms have the same chance of match scheduling. As a result, we focus more on the following two locally run experiments.

Average human evaluation: As highlighted in the Settings section, we have obtained all training data of the New York Times RPS Applet [3]. Since the dataset contains 200,000 rounds of playing history of humans vs the applet, we build a program named `AverageHuman.py` to generate "typical" moves of an "average" human. Therefore, the result of our algorithms vs this `AverageHuman.py` is a reasonable indicator about how hard the algorithms can beat human players on average. In the poster session, we invited some people to play with our algorithms with the demo web page and found that the results can roughly match that of `AverageHuman.py`.

We consider all algorithms described above, including Random, HMM, RNN, LR, SVM, RL_table, RL_approx, RL_features and RC. For each algorithm, we run it against `AverageHuman.py` for 100 rounds and further repeated it for 10 times. All the results are organized as a box plot in Fig. 7.

Ranking (out of 40)	Program	Winning Rate (out of 1170 games)
11	RL_features	74.3%
19	RL_table	54.3%
26	RC	49.0%
29	Random	47.1%
34	RL_approx	30.9%
36	AverageHuman	22.3%
37	SVM	12.1%
38	RNN	11.2%
39	LR	9.6%
40	HMM	5.8%

Figure 8: Local tournament results

As seen in Fig. 7, all algorithms of sequence modeling and behavior cloning (*i.e.*, HMM, RNN, LR, SVM) perform nearly perfectly. This is anticipated, as the pattern of `AverageHuman.py` is fixed. What is interesting is that the performances of all reinforcement learning algorithms, except that of RL_approx, are far from ideal. This is due to the capacity of state representations of different algorithms. For RL_table, since it uses $N = 4$ pairs of player-computer gestures as the state, it can only represent a total of 9^4 configurations. Similarly, RL_features have only 3 pairs of gestures and the capacity is 9^3, which is even smaller. In contrast, since RL_approx applies function approximation, we can use $N = 10$ pairs of gestures to approximate the state, which is capable of capturing more configurations. Finally, since RC combines SVM and RL_features, its performance also lies between those of the two algorithms.

Local tournament evaluation: We crawled top 30 programs on `RPSContest` leaderboard, put them together with all our algorithms and ran 3 local tournaments. During each tournament, every program plays 10 matches against every other program and the final results are aggregated. This task is difficult, since we only compare ours with some of the best programs publicly available on the internet. It is also more stable than `RPSContest` leaderboard, since we ensure that the number of matches for each program is the same and the winning rates do not favor most recent victories.

Experimental results averaged on all 3 tournaments are concluded in Fig. 8. Note that sequence modeling and behavior cloning algorithms (HMM, RNN, LR, SVM) have poor performance against top 30 programs, which is anticipated since adversarial programs can typically change their patterns instead of using a fixed one. It is interesting to see that `AverageHuman.py` itself also ranks near the end.

Unlike other reinforcement learning algorithms, RL_approx does not work well due to the limit of linear approximation to the Q function, as indicated before. RL_table and RL_features have more reasonable state representations and are less prone to long term patterns, which is beneficial for winning competitions. RL_features was even ranked 11^{th} among the best programs.

As expected, RC have a performance somewhere between that of RL_features and SVM.

5 Conclusion

We formulated the game of rock-paper-scissors as a machine learning problem and tried algorithms of sequence modeling, behavior cloning and reinforcement learning. In order to compare performance of different algorithms, we also proposed novel evaluation approaches. In the experimental part, we extensively studied the pattern matching ability using the New York Times Applet playing

history and the ability of combating adversaries using highest ranked algorithms on RPSContest leaderboard. A reinforcement learning algorithm with carefully hand-crafted features of states have once won the 5th place of 2289 programs on the leaderboard. Inspired by the findings of the experiments, we proposed to combine two of the algorithms with a multi-armed bandit method called reinforcement comparison, and showed that it can interpolate the performance in both evaluation settings.

All the algorithms are implemented in Python 2.7. We also created a dynamic web page with JavaScript and Python framework Flask to provide a funny environment for humans to play with the algorithms. All codes can be found in the attached zip file.

Acknowledgments

We sincerely thank Shawn Bayern and Tom Denton for providing the New York Times Applet dataset. We also greatly appreciate the help of TSAIL and Stanford AI Lab for letting us use their computer clusters.

References

[1] Neil Farber. The Surprising Psychology of Rock-Paper-Scissors. https://www.psychologytoday.com/blog/the-blame-game/201504/the-surprising-psychology-rock-paper-scissors, April 2015.

[2] Douglas Walker and Graham Walker. *The official rock paper scissors strategy guide*. Simon and Schuster, 2004.

[3] Rock-paper-scissors: You vs. the computer. http://www.nytimes.com/interactive/science/rock-paper-scissors.html, October 2016.

[4] Tristan Breeden. CS229 Final Report: RPS Bot Discerning Human Patterns in a Random Game, December 2015.

[5] Gabriele Pozzato, Stefano Michieletto, and Emanuele Menegatti. Towards smart robots: Rock-paper-scissors gaming versus human players. In *PAI@ AI* IA*, pages 89–95. Citeseer, 2013.

[6] Diego FP de Souza, Hugo CC Carneiro, Felipe MG França, and Priscila MV Lima. Rock-paper-scissors wisard. In *2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence*, pages 178–182. IEEE, 2013.

[7] Byron Knoll, Daniel Lu, and Jonathan Burdge. RPSContest. http://www.rpscontest.com/, October 2016.

[8] The Second International RoShamBo Programming Competition. http://webdocs.cs.ualberta.ca/~darse/rsbpc.html, March 2001.

[9] Rps-25. http://www.umop.com/rps25.htm.

[10] Volodymyr Kuleshov and Doina Precup. Algorithms for multi-armed bandit problems. *arXiv preprint arXiv:1402.6028*, 2014.

[11] Armin Ronacher. Flask, web development, one drop at a time. http://flask.pocoo.org/, 2016.

[12] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.