DeepShuai: Deep Reinforcement Learning based Chinese Chess Player

Chengshu Li [1] **Kedao Wang** [1] **Zihua Liu** [1]

Abstract

Chinese chess has long been viewed as one of the most popular board games in China. It has a larger state and action space than chess, hence greater difficulty for AI to conquer. Many previous work focused on search based algorithm or simple TD learning to tackle Xiangqi. However, in this project, we propose a deep reinforcement learning based algorithm inspired by AlphaGo. We first used supervised learning to initialize player agent and use reinforcement learning algorithms to update the players against a commercial Xiangqi agent called Elephant Eye. We are able to achieve a consistent 56% win rate over Elephant Eye evaluated in 100 games.

1. Introduction

Xiangqi is a traditional chess game much like chess itself with unique pieces and different rules to move these peieces. In terms of game tree complexity, Xiangqi surpasses Chess with a branching factor of 38 and game tree complexity of 150 compared to branching factor of 35 and game tree complexity of 123 to Chess. The board for Xiangqi is also larger: a 10×9 board with 17 pieces on each side. In addition, there are also additional restraints and conditions imposed on the way Xiangqi pieces move. For instance, the King equivalent of Chess in Xiangqi cannot leave a set ”Palace”, a 3×3 grid in the middle of each side. Differences aside, there are significantly less research on autonomous player for Xiangqi. Most existing researches explore search based methods and evaluate against a commercial agent Elephant Eye. In this project, we propose a deep method of encoding game states and devising policies to play Xiangqi inspired by methods and principles employed by AlphaGo.

The following paper is structured as follow: Section 2 will introduce some current research into Xiangqi Agent. Section 3 will introduce our dataset and environment. Section 4 and 5 will discuss approaches we have experimented and their experimental results. Section 6 and 7 will offer insights on some challenges and future work and a conclusion.

2. Related Work

Early reinforcement learning based agent for board games such as Chess or Xiangqi first originated in TD-Gammon by Tesauro (Tesauro, 1995). In his paper, he first used a neural network to approximate the value of a board state and applied TD learning on the game Backgammon, a popular board game at the time. Inspired by Tesauro’s work, Yin et. al first proposed applying Temporal Difference learning on Xiangqi (Yin & Fu, 2012). Current day Xiangqi research focuses on primarily two different categories: 1) advancing search based algorithms, 2) new state evaluation functions. An example of former category of research is by Liu et. al, who devised a variation of alpha-beta pruning for Xiangqi aimed at learning state search at end games (Liu & Guo, 2012). On the other hand an example of latter category is by Fu et. al, who applied a three layer feed forward neural network, combined with information from prior heuristic, to obtain a new evaluation function for the value of a position (Fu & Yin, 2012). However, with the rise of deep learning methods, we now have tools to encode more complex information about the board state. Such methods have been employed in other games like Chess or Go. Lai first applied deep reinforcement learning to produce Giraffe. In this work, a multi-layered perceptron network was used to encode the value of a board state to perform TD learning (Lai, 2015). A more recent and highly impactful work is AlphaGo by DeepMind. In this work, a deep convolutional neural network is used to approximate both policy and value of a game board (Silver et al., 2016).

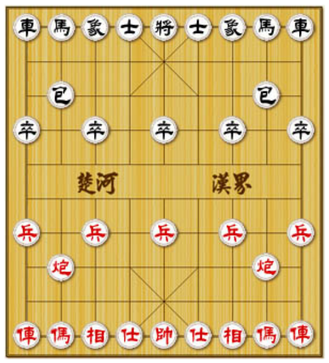


Figure 1. Example Xiangqi Board

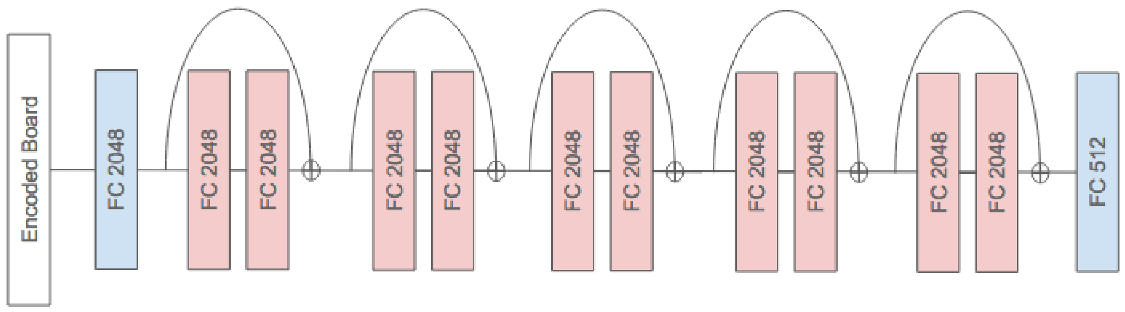


Figure 2. Shared Network Structure: 5-layer Residual Network

In our project, we will closely follow strategies employed by AlphaGo: we will first use supervised training followed by reinforcement learning.

3. Dataset

3.1. Dataset

To complete the aforementioned task, we have scraped 70,000 complete expert games from a database for Xiangqi, each with on average of 100 moves. Each move is in Xiangqi notation style. We have also built up an environment Xiangqi environment that allows us to virtually replay these 70,000 games to produce 5,595,966 unique board state without the draw games. We used this dataset in the supervised setting by splitting the dataset into 80% of training data and 20% of validation data. To prevent the network from memorizing the game state itself instead of evaluating the board, we split the dataset based on game: board states from one game is either entirely in train set or validation set.

Results	Percentage
Red win	37.78%
Black win	27.90%
Draw	34.32%

Table 1. Distribution of Game Results in Scrapped Dataset

3.2. Environment

As mentioned before, we have implemented a chinese chess environment that will be core of our agents interaction with the opponent. The use for this environment is as followed. 1) We use this environment to generate the dataset from Xiangqi Notation to independent board positions that we use to perform supervised learning. 2) We use this environment to generate all possible next board states following the current one for evaluating each board state to determine the next state with highest value via our network in the reinforcement learning setting. 3) Integrate with Elephant Eye (Eleeye), a commercial Xiangqi agent. And 4) Allows the network to self-play to learn the end-game scenarios.

4. Approach

In this section, we will discuss in details three methods we experimented to create a Xiangqi agent. These approaches include value based method, policy based method, and actor-critic method. In the value based method, our network approximates the value of the given state based on each player; in the policy based method, our network instead predicts the policy, namely the next move taken by the player, given the board state; in actor-critic method, a network predicts the next move and approximates an advantage estimate on the predicted policy. The network architecture for these methods are very similar. Each of these networks takes a vectorized, one-hot encoded board state as input. They also share the same stem of network and only differ by the last few output layers. The shared portion is a 5-layer Residual Network with 2048 hidden neuron per layer as demonstrated in Figure 2. No convolution layer is used in any methods.

Each of the method can be split into three distinct phases: supervised learning, reinforcement learning with Eleeye, and reinforcement learning through self-play. The first stage of the three is supervised learning. Depending on the method, we can create training criteria on each of the 5,595,966 unique game positions scrapped from the Xiangqi database. The point of this supervised learning stage is to bootstrap reinforcement learning with some prior information learned from expert moves. The second stage of the three is reinforcement learning by playing against Eleeye. In this stage, our network will be updated with both our experiences playing against Eleeye and their experience from play against us. The last stage is self-play. In this stage, our agent will play against a past variant of itself to further increase the strength of its policies. In the following subsections, we will discuss in detail each method we experimented.

4.1. Value Based Method

4.1.1. Supervised Learning

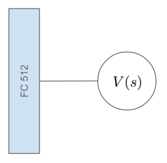
We trained the value network via supervised learning, followed by Temporal Difference learning. We trained a deep residual value network that takes board positions as input, and outputs a single value between -1 to 1, with -1 being a lost game, 0 being a draw, and 1 being a win. The input is the chess board encoded as a 1-D one-hot vector, with dimension being $board\_height * board\_width * n\_pieces = 10 * 9 * 17 = 1530$. The pieces dimension encodes all pieces from both players (`King, Assistant, Bishop, Knight, Rook, Cannon, Pawn, Pawn_across_river`), as well as empty cell, for a total of $8 + 8 + 1 = 17$ piece types.

We doubled the training data by switching perspective between red and black players. For example, a red players board position corresponding to a win, would become a black players position corresponding to a loss when the board is rotated 180 degrees. The labels are discounted sum of future rewards:

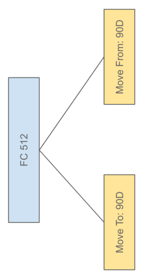
$$R = \sum_t \gamma^t z \tag{1}$$

$$z = \begin{cases} +1 & win \\ 0 & draw \\ -1 & loss \end{cases} \tag{2}$$

Where reward is only non-zero at end of game (z). By using discounted sum of future rewards as training label, the network can prioritize value over states closer to a terminal state.



(a) Value Net: output layer



(b) Policy Net: output layer

4.1.2. Reinforcement Learning

We subsequently trained the RL agent via reinforcement learning, by initializing the same value network with weights learned from supervised learning. We trained the RL agent via TD learning, by minimizing the following objective:

$$\mathbb{E}\left[r(s,a) + \gamma V(s') - V(s)\right] \tag{3}$$

Where the transition is deterministic $s' \leftarrow s, a$. Each action is chosen with e-greedy probability epsilon. At each step, we generate all legal next states of the chess board, feed them through the value network to obtain the values for all next states. With probability 1 - ϵ the agent picks the action yielding the highest value for next state. With probability ϵ the agent chooses an action by random. ϵ is a hyperparameter.

We used double Q-learning (Van Hasselt et al., 2016) as the reinforcement learning algorithm . Double Q-learning works by using two sets of weights, one for the Q-network and one for the target network. The two networks are swapped with some probability (we chose 50%) after a fixed number of iterations.

Experience replay is used to break the temporal dependency between actions. Experience replay helps the network converge faster in situations where the reward signal is sparse (Lin, 1993), for instance when an agent only receives award at the end of the game. Using naive TD learning with exploration, it would take an unreasonable amount of trials to propagate the reward through the early states. We allocated an experience buffer to store `(state, action, next_state, reward)` tuples. At backpropagation time, we sample a batch of experience tuples from the buffer to update the weights of network.

4.2. Policy Based Method

4.2.1. Supervised Learning

We then formulated the problem from a policy perspective. A policy network takes state as input and outputs action probabilities. In the domain of Chinese Chess, a brute force way to list all possible moves is huge, due to the large product of grid cells, piece types, and move options. Therefore we chose an efficient output space of two vectors, each with size $board\_width \times board\_height = 10 \times 9 = 90$ (spa). One vector represents the from-coordinate of a piece, while the other vector represents the to-coordinate of a piece. The policy network takes one-hot encoding of chess board as input. We trained the policy network with the scraped state - action pairs.

4.2.2. Reinforcement Learning

During reinforcement learning phase, we used the learned network from supervised learning phase to train the agent.

When taking a step, we feed the current board state into the network. The network outputs two probability distribution over from-coordinates and to-coordinates. We calculate the joint probability of all legal moves, and select the move with the highest joint probability:

$$P_{move} = P_{from} \cdot P_{to} \tag{4}$$

During reinforcement learning, we used vanilla policy gradient to train our agent.

$$\bigtriangledown_\theta J(\theta) = \mathbb{E}\left[\bigtriangledown_\theta \, log \, \pi(s,a) A(s,a)\right] \tag{5}$$

Where the advantage is

$$A = \sum_t \gamma^t z \tag{6}$$

We initially played our agent against Elephant Eye, an open-source Chinese Chess AI. Since our agent initially loses all the games against Elephant Eye, experience from both the RL agent and Elephant Eye are used for TD learning. The intuition is that, by providing balanced positive (winning) and negative (losing) examples, our network is able to learn good behavior, and at the same time avoid bad behavior.

4.3. Actor-Critic Method

Policy gradient is prone to variance, where a tiny policy gradient step in the wrong direction could result in a disastrous policy useless for learning. In order to reduce the variance of policy updates, we used Actor-Critic learning to train our RL agent, by initializing the policy network and value network using weights learned from supervised learning phase.

Actor-Critic method works by using one policy network to predict the actions, and another network to predict the value of a state. The policy network is trained via policy gradient:

$$\bigtriangledown_\theta J(\theta) = \mathbb{E}\left[\bigtriangledown_\theta \, log \, \pi(s,a) A(s,a)\right] \tag{7}$$

The value network is trained via TD learning:

$$min \left\{\mathbb{E}\left[r(s,a) + \gamma V(s') - V(s)\right]\right\} \tag{8}$$

We used Generalized Advantage Estimation (Schulman et al., 2015), where advantage is defined as:

$$A_t^{GAE} = \sum_l (\gamma\lambda)^l \delta_{t+l} \tag{9}$$

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{10}$$

We experimented with two network architectures:

- Separate networks: we used two separate networks, one for policy network and one for value network.
- Shared network: we used a single network with shared bottom layers, and outputs both action probabilities and state value. The hope here is to use multi-task learning to help the network transfer knowledge between two tasks.

Both architectures are first trained via supervised learning, and then via actor-critic reinforcement learning.

5. Experiment

In this section, we will present experimental results for all three different methods in both supervised learning stage and reinforcement learning stage. All of our experiments are trained on a single NVIDIA GeForce GTX TITAN GPU.

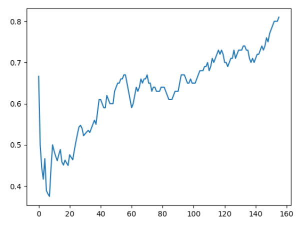
5.1. Value Based Method

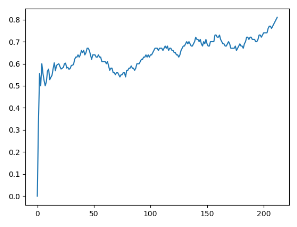
5.1.1. Supervised Learning

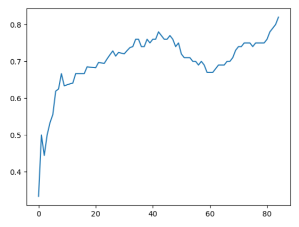
For training a value network, as suggested by the previous section, the output of our value network is a regression value. After 6 epochs of training, we are able to achieve a validation loss of 0.1877 with discount factor $\gamma = 0.98$. Droppout of 0.5 is used whereas no batch normalization is applied. For reference, we employed the same kind of value evaluation scheme with that of AlphaGo where they achieved a validation loss of 0.23.

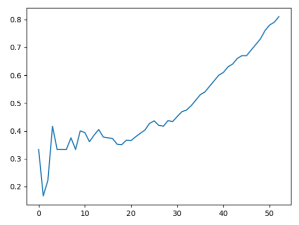
5.1.2. Reinforcement Learning

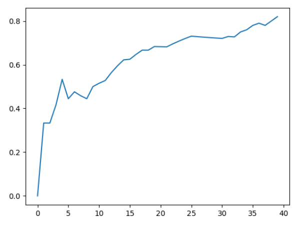
For the reinforcement learning section, the performance of TD learning proves to be less than satisfactory. We applied TD learning with experience replay on both Agent's experience and Eleeye's experience. After 7000 epochs of 2500 games each of playing against Eleeye, Value Network is able to reduce the error objective of Q-learning down to 0.05. We experimented multiple representation of the board: both one-hot encoding of all pieces and direct representation of the board given a heuristic value of each piece. However the win rate against even the easiest Eleeye setting is still 0 evaluated over the past 100 games after 7000 epochs of training.


(c) Generation 1


(d) Generation 2


(e) Generation 3


(f) Generation 4


(g) Generation 5

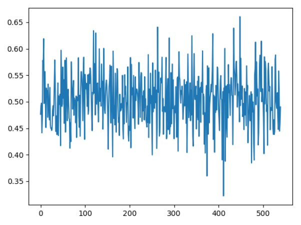

(h) Generation 6

Figure 3. Win rate of self-play over time across generations

5.2. Policy Based Method

5.2.1. Supervised Learning

For training a policy network, the output of our network becomes two vectors of length 90 to encode the broad position that the player should move from and move to. After 11 epochs of training, we are able to achieve a validation accuracy of 46.29% on move from position and 60.33% on move to position. Both batch normalization and dropout keep probability of 0.4 is used. For reference, AlphaGo got a 55% validation accuracy on supervised training of policy network, with only one output probability because each stone and position are encoded equivalently in Go. Training graph for supervised learning is provided in Figure 4.

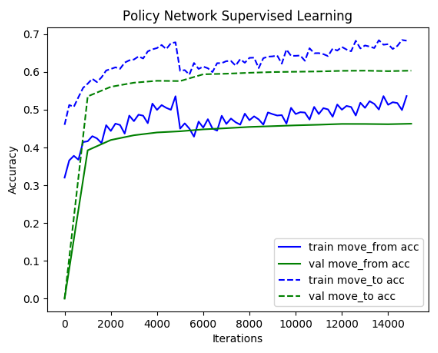


Figure 4. Policy Network Accuracy over time

5.2.2. Reinforcement Learning

For the reinforcement learning stage, agent makes decision based on the max probability legal move paired with REINFORCE algorithm to update the policies. However, once again, playing against Eleeye does not offer us much gain. The our win rate against Eleeye is kept consistent at 0%.

On the other hand, we are able to observe some results from self-play. For self-play, we have two agents initialized to policy network from supervised training, but only one agent is received gradient updates. We call a generation to be an extended period of time before either 2000 epochs or the updated agents has a 80% win rate over the un-updated agent. Our observed result was that the agent quickly learns from playing against itself. About after 5 generations, the win rate of updated agent plateaus at 50% at generation 6. Win rate with respect to training epochs are shown for each of the 6 generations in Figure 3. However, when we evaluate the agent against Eleeye, our win rate is 1%. When we manually examine each steps our agent makes, we found out that the agent is able to learn some standard opening moves as well as taking and defending crucial pieces.

5.3. Actor-Critic Method

5.3.1. Supervised Learning

For using actor critic method, we combine the value network and policy network from both previous method. The output of our network is both the single dimension regression value and two position vectors of length 90. After 13 epochs of training, we are able to achieve a validation loss of 0.192 on value regression and validation accuracy of 42.49% on move from position and 56.91% on move to position on policy prediction. Both batch normalization and dropout of 0.4 is used. Training graph is provided in Figure 5.

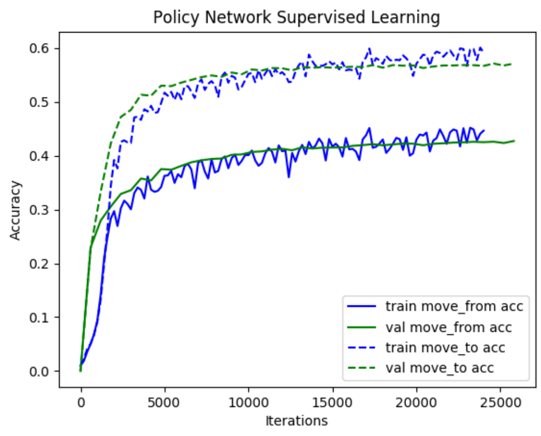


Figure 5. Policy-Value Joint Network Accuracy over time

5.3.2. Reinforcement Learning

Very similar to policy based method, for the reinforcement learning stage, agent makes decision based on the max probability legal move. In order to explore the state and action space enough, we decided to sample action based on the softmax probability that our policy network produces rather than just taking an argmax. REINFORCE algorithm is once again used but with a twist. We employed many different techniques to both reduce variance and ensure propagation of information. We also use generalized advantage estimate – we calculate the advantage function through the value output of the state compared to the immediate reward plus a discounted value of the resulting state. This advantage function is used to compute the gradients to further reduce variance that we encounter.

Through this method, we are able to achieve great results: after around 700 epochs of 2500 games each, we are able to achieve a win rate of consistently 56% against Eleeye. Although this version of Eleeye is not the strongest version of the Xiangqi agent, we are able to achieve a non-trivial win rate. After careful observations, we are able to see that even in games that DeepShuai draws with Eleeye, DeepShuai is able to make sensible moves.

6. Challenges and Future Work

The main challenges that we face throughout the project is on the reinforcement learning part. We achieved close to state-of-the-art results in the supervised learning phase of all three methods. Yet the RL results for the first two methods are less than satisfactory.

For our RL models, training and validation loss steadily decrease but we fail to transfer decreasing loss into actual win games. For our actor-critic method, we did achieved over 50% win rate. Yet after reading the log, we realized that we almost always won the game in the exactly same way. The reason is that Elephant Eye is a deterministic AI agent and we also only won the game if our agent played first. As training progressed, we failed to further increase our win rate and started to overfit to this particular opponent. We have tried to use e-greedy to add randomness to the system to see if our agent can beat Elephant Eye if Elephant Eye plays first. We haven't achieved good results as the model quickly diverged to 0 win rate again.

Another challenge is that since we lose most of the games to Elephant Eye in the reinforcement learning phase, our agent basically "unlearns" what it has learned from supervised learning, especially for the first two methods, because it decreases the probability of its action for a lost game. Furthermore, we found policy gradient method relatively unstable and we experienced multiple loss and gradient explosion when training REINFORCE.

For future work, we would love to try Monte Carlo tree search (MCTS) on top of our value network to have a more accurate heuristic. We also want to introduce trust region policy optimization (TRPO), which has proven to be more stable than the vanilla policy gradient method. Furthermore, we need to develop a better exploration mechanism for actor-critic system when playing against a deterministic AI to avoid overfitting. Lastly, we want to try Asynchronous Actor-Critic to speed up our training time.

7. Conclusion

In conclusion, in this study, we proposed and experimented several deep reinforcement learning based approach for an autonomous Xiangqi agent, including Value-based network with TD learning, Policy-based network with REINFORCE, and actor-critic-based network with REINFORCE. Among them, actor-critic based method significantly out performs other variants and achieved 56% win rate against Eleeye. This study shows that Xiangqi, much like Go or Chess, can be tackled with non-traditional methods other than search. This work also opens up the gate to many future work, for we have also mapped out many potential pitfalls and problems future research may encounter in attempting to create stronger agents.

8. Acknowledgment

Throughout this project, all team members contributed equally in designing the model, running experiments, and drafting the report, milestone and poster.

References

Spawk.fish. URL http://spawk.fish/posts/2016/02/policy-network/.

Fu, Tingting and Yin, Hongfeng. Designing a hybrid position evaluation function for chinese-chess computer game. In *Software Engineering and Service Science (ICSESS), 2012 IEEE 3rd International Conference on*, pp. 75–78. IEEE, 2012.

Lai, Matthew. Giraffe: Using deep reinforcement learning to play chess. *arXiv preprint arXiv:1509.01549*, 2015.

Lin, Long-Ji. *Reinforcement learning for robots using neural networks*. PhD thesis, Fujitsu Laboratories Ltd, 1993.

Liu, Hal-Tao and Guo, Bao-En. A new pruning algorithm for game tree in chinese chess computer game. In *Machine Learning and Cybernetics (ICMLC), 2012 International Conference on*, volume 2, pp. 538–542. IEEE, 2012.

Schulman, John, Moritz, Philipp, Levine, Sergey, Jordan, Michael, and Abbeel, Pieter. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

Silver, David, Huang, Aja, Maddison, Chris J, Guez, Arthur, Sifre, Laurent, Van Den Driessche, George, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Tesauro, Gerald. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

Van Hasselt, Hado, Guez, Arthur, and Silver, David. Deep reinforcement learning with double q-learning. In *AAAI*, pp. 2094–2100, 2016.

Yin, Hong-Feng and Fu, Ting-Ting. Applying temporal difference learning to acquire a high-performance position evaluation function. In *Computer Science & Education (ICCSE), 2012 7th International Conference on*, pp. 80–84. IEEE, 2012.