Learning to Play Minichess Without Human Knowledge

Karthik selvakumar Bhuvaneswaran
karthik0@stanford.edu

Abstract

Implementing a self play based algorithm using neural networks has become popular after the tremendous success of Alpha Zero by Deep Mind in the game of go. Replicating the results for a game with smaller search space like TicTacToe, Connect4 etc has already been proven in alpha-zero-general (Surag Nair et al, 2017), but for games with larger search space like chess requires scaling. In this paper we apply scaled up version of alpha-zero-general for the game of Minichess and evaluate our learning algorithm with random and greedy baselines.

1 Introduction

It was estimated that games like Go, which have a large branching factor, and where it is very difficult to determine the likely winner from a non-terminal board position, would not be solved for several decades. However, AlphaGo (Silver et al. 2016), which uses recent deep reinforcement learning and Monte Carlo Tree Search methods, managed to defeat the top human player, through extensive use of domain knowledge and training on the games played by top human players. Many of the existing approaches for designing systems to play games relied on the availability of expert domain knowledge to train the model on and evaluate non-terminal states. Recently, however, AlphaGo Zero (Silver et al. 2017b) described an approach that used absolutely no expert knowledge and was trained entirely through self-play. In our work, we extract ideas from the AlphaGo Zero paper and apply them to the game of Minichess (5X5). For evaluation, we compare our trained agents to random and greedy baselines.

2 Related Work

DeepMind published a paper of Alpha Zero on arXiv that extends AlphaGo Zero methods to Chess and Shogi. However, the code is not open source and will not be released by DeepMind. Moreover, they use about 5000 TPUs to generate games in parallel. This makes it very difficult to replicate the results and extend the work to other games. Alpha Zero General is open source single-thread single-GPU version that works for any game and any framework (currently PyTorch, TensorFlow and Keras). It works quite well on small games such as 6x6 Othello after 2-3 days of training on a single GPU. We will be extending Alpha Zero General project by analyzing the components that can be parallelized and training a Neural Network for the game of Minichess.

3 Methods

We provide a high-level overview of the algorithm we employ, which is based on the AlphaGo Zero (Silver et al. 2017b) paper. The algorithm is based on pure self-play and does not use any human knowledge except knowing the rules of the game. At the core, we use a neural network that evaluates the value of a given board state and estimates the optimal policy. The self-play is guided by a Monte-Carlo Tree Search (MCTS) that acts as a policy improvement operator. The outcomes of each game of self-play are then used as rewards, which are used to train the neural network along with the improved policy. Hence, the training is performed in an iterative fashion the current neural network is used to execute self-play games, the outcomes of which are then used to retrain the neural network. The following sections describe the different components of our system in more detail.

3.1 Gameplay

A 5x5 board is the smallest board on which one can set up all kind of chess pieces as a start position. We consider Gardner's minichess variant in which all pieces are set as in a standard chessboard (from Rook to King). This game has roughly $9 * 10^{18}$ legal positions (Mehdi

Mhalla et al, 2013) and is comparable in this respect with checkers.

Figure 1: Popular Minichess layouts (5X5)

3.2 Neural Network Architecture

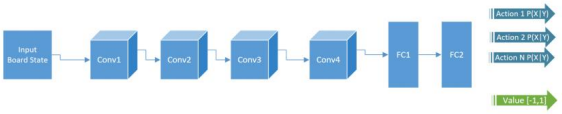


Figure 2: Neural Network Architecture

We use a neural network parameterized by θ that takes as input the board state s and outputs the continuous value of the board state $v_\theta \epsilon$ [-1,1] from the perspective of the current player, and a probability vector $\vec{p}$ over all possible actions. $\vec{p_\theta}$ represents a stochastic policy that is used to guide the self-play. The neural network is initialized randomly.

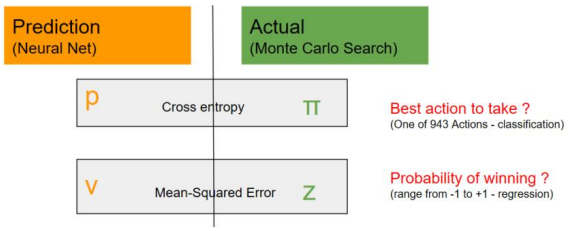


Figure 3: Parameters in Neural Network and choice of loss functions

At the end of each iteration of self-play, the neural network is provided training examples of the form $(s_t,\vec{\pi_t},z_t)$. $\vec{\pi_t}$ gives an improved estimate of the policy after performing MCTS starting from st, and $z_t \epsilon$ [-1,1] is the final outcome of the game from the perspective of the current player.

The neural network is then trained to minimize the following loss function:

$$l = -\sum(v_\theta(s_t) - z_t)^2 + \vec{\pi_t} log(\vec{p_\theta}(s_t))$$

We use a neural network that takes the raw board state as the input. This is followed by 4

Hyperparameter	Value
MCTS Simulations	200
Exploration (cpuct)	1
Learning Rate	0.0005
Update Threshold	0.5
Arena Compare	20
Iterations	100
Episodes	100
Data Augmentation	Two way Symmetry
Value Activation	tanh
Policy Activation	Softmax
Batch Size	128
Number of Layers	7 (4 CONV, 3 FC)
Regularization	Dropout (0.2)
Optimizer	Adam
Normalization	Batch Normalization

Figure 4: Hyperparams used for Preprocessing and Training

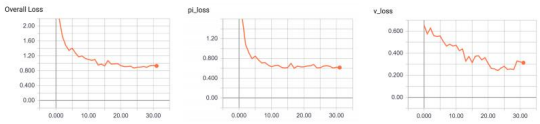


Figure 5: Loss values after each epoch

convolutional networks and 2 fully connected feed forward networks. This is followed by 2 connected layers - one that outputs v_θ and another that outputs the vector $\vec{p_\theta}$. We tried standard Neural Net with 6 layers, but spatial information was lost and performance was bad. Training is performed using the Adam (Kingma and Ba 2014) optimizer with a batch size of 128, with a dropout (Srivastava et al. 2014) of 0.2, and batch normalization (Ioe and Szegedy 2015). The neural net is implemented in Keras with tensorflow-gpu backend.

3.3 Distributed Architecture

We ran our experiments on AWS EC2 instances, but architecture is generic for all cloud services. Following table explains the different components and type of instance created in AWS.

Component Name	Instances
Training Data Generator	4 x p2.xlarge
Training Data Storage	1 x AWS S3
Neural Net Trainer	2 x p2.4xlarge
Pitter	1 x p2.xlarge

Table 1: Distributed Arch Components

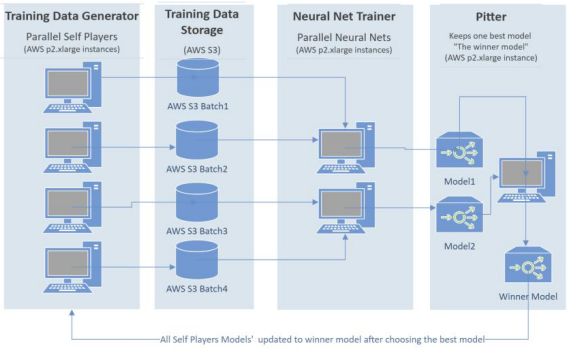


Figure 6: Distributed Architecture in AWS

3.4 Dataset used - Data Generation through pure Self Play

Training Data Generator component is responsible for carrying out self plays and generating dataset required for training our Neural Network. It creates two players and plays the game till the end state (Won/Lost/Draw). On every move, it collects three data points: board state, policy output and value output of the network. This dataset is then uploaded to AWS S3 which will be consumed by *Neural Net Trainer*.At any point of time, currently known best model is used and the network becomes smarter and smarter after training for more iterations with larger training data. The dataset is shuffled to avoid bias between two consecutive moves and data augmentation is carried out by flipping the board from right to left. Castling and en passant were not considered to simplify the scope.

3.5 Neural Net Trainer

Neural Net Trainer component performs the training and needs to be more powerful to crunch the enormous amount of batch data generated by self play and perform several iterations of gradient descent. Hence it is chosen to be four times powerful than *Training Data Generator* in terms of number of physical GPU hardware and CPU concurrency. This component performs training and outputs the model with lower loss.

3.6 Pitter and best model selection

Pitter component takes all the models from *Neural Net Trainers* and performs pitting. Depending on the winner, the best model is decided and all the *Training Data Generators* are updated with this latest model, which helps in improving the quality of Training data being generated, eventually leading to an expert model.

3.7 Training Performance

- We evaluated the training till 30 iterations in both single instance and the proposed distributed architecture.
- We observed 2.5 times improvement in training speed with distributed setup over single instance (ref Figure 7).

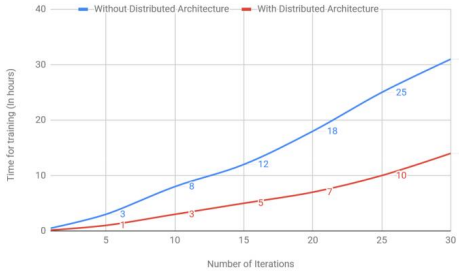


Figure 7: Single Instance vs Distributed Architecture

4 Experiments

We ran experiments on a 5X5 version of Chess referred as Minichess. We used 200 MCTS simulation per move which is good enough simulation for a 5X5 board. With 100 episodes per training iteration and 100 iterations of training it took 8.5 hours of simulation.

Inorder to scale the training, any MCTS search beyond 200 search deep will be considered as draw. This happened in scenarios when there is only White King and Black King left on the board and both players cannot bring the game to an end. This draw state is tailored for subset of board games like Minichess and might lead to erroneous prediction for other board games where the valid simulation might itself be more than 200 level deep.

4.1 Baselines

We implemented three baselines for comparison with our trained AI Player.

- The first is a *random player* baseline. A random player chooses from one of the

valid moves randomly at each step in the game.

- The second is a *greedy player* which chooses a move that causes the attack which maximizes the score. For computing score each Piece is assigned a weight.

Piece	White Weight	Black Weight
Pawn	100	-100
Knight	280	- 280
Bishop	320	-320
Rook	479	-479
Queen	929	-929
King	6000	-6000

Table 2: Weights of Pieces on Greedy Player

- Third is comparison of *different iterations of the model* itself with the best model.

4.2 Results

The results of the different baselines are below:

Baseline	Color	Won	Lost	Draw
Random Player	White	10	0	0
	Black	10	0	0
Greedy Player	White	10	0	0
	Black	3	0	7
Model Version 1	White	10	0	0
	Black	7	0	3
Model Version 5	White	10	0	0
	Black	0	0	10
Model Version 15	White	0	0	10
	Black	0	0	10

Figure 8: Results of pitting Best Model (V21) against other players

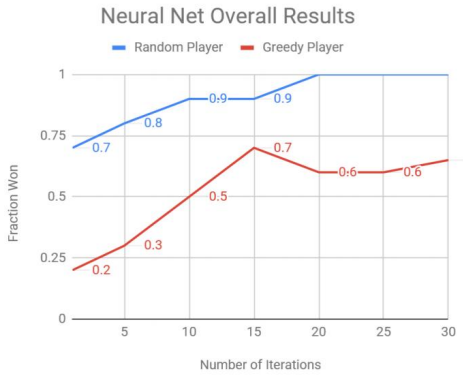


Figure 9: Overall Performance of Neural Net over other baselines

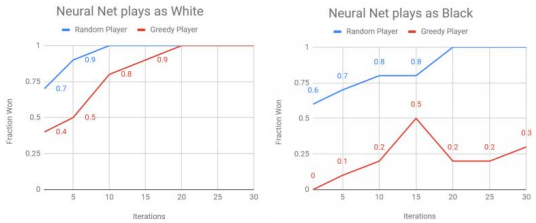


Figure 10: Neural Net as white vs black

4.3 Discussion

After 21 iterations, when evaluated:

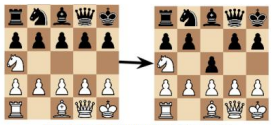
- Our model defeats random player 100% of the times, irrespective of whether it plays as Black or White (ref Figure 10).
- Defeats greedy player 100% when Neural Net takes first turn (White), but has lower winning rate when it plays as Black (ref Figure 10).
- The best model never loses any game against any baseline, so it manages to either win or draw the game (ref Figure 8).
- When Neural Net pits against Random and Greedy baselines on entirely new board layouts like BabyChess and Mallet, produces impressive results with very few losses (ref Figure 11).

Chess Layout	Baseline	Color	Won	Lost	Draw
BabyChess	Random	White	10	0	0
		Black	9	1	0
	Greedy	White	10	0	0
		Black	0	10	0
Mallot	Random	White	10	0	0
		Black	9	1	0
	Greedy	White	10	0	0
		Black	10	0	0

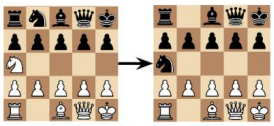
Figure 11: Trained on Gardner and transferred to other layouts

5 Observations

- While pitting against Iteration 1 of the Model, when we chose white and placed Knight, Neural Net doesn't attack and instead plays a passive Pawn forward move.
- It becomes more aggressive after iteration 5 and attacks the Knight with its Knight.

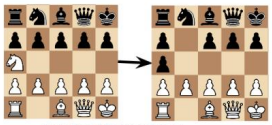


Iteration 1: Black plays passively, ignores to attack the Knight.

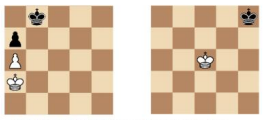


Iteration 5: Black plays aggressively, but plays with Knight instead of Pawn.

- After iteration 15, Neural net uses Pawn for attack if possible rather Knight to mitigate the loss in the next move.
- At higher iterations most of the game ends in draw, with either King defending its Pawn or only two kings left on the board as shown in the board.



Iteration 15: Black plays aggressively, and uses Pawnt.



Iteration 30: During self play at higher iterations 30+, most games ends in Draw

6 Conclusion

- Trained model beats the random, greedy baselines and performs decently on other layouts.
- Monte Carlo Tree Search and CNN can approximate search space as large as $9 * 10^{18}$.
- Parallelizing self play, training and pitter by leveraging cloud services improves the performance substantially.

7 Future Work

Evaluate against human and minimax baselines like stockfish. MCTS was the major bottleneck during training, leverage asynchronous CUDA version of MCTS to accelerate the training further. Evaluate on larger games like chess or go with a significantly bigger distributed setup. Analyze the transferablity of the model from one game to another like:

- Extend 5X5 chess model to work on 8X8 board
- Extend 8x8 chess model games with similar rules and strategies like chess
- Analyze the model to games with entirely different rules like Othello.

8 Code

- Github repository can be found at https://github.com/karthikselva/alpha-zero-general.
- Short walk through video can be found at https://youtu.be/NxzABCCzYCE.

Acknowledgment Thanks to Patrick Cho for his direction and excellent mentorship and rest of the staff for helping us understand the architecture behind the beautiful AlphaZero.

References

[Mehdi Mhalla, Frederic Prost 2013] Gardner's Minichess Variant is solved. *arXiv e-print (arXiv:1307.7118)*

[Surag Nair, Shantanu Thakoor, Megha Jhujhunwala 2016] Learning to Play Othello Without Human Knowledge *github.com/suragnair/alpha-zero-general*

[Silver et al. 2017a] Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *ArXiv e-prints.*

[Browne et al. 2012] Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games 4(1):143.*

[Heinz 2001] Heinz, E. A. 2001. New Self-Play Results in Computer Chess. *Berlin, Heidelberg: Springer Berlin Heidelberg. 262276.*

[Ioe and Szegedy 2015] Ioe, S., and Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *In International Conference on Machine Learning, 448456.*

[Kingma and Ba 2014] Kingma, D., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980.*

[Srivastava et al. 2014] Srivastava, N.; Hinton, G. E.; Krizhevsky, A.; Sutskever, I.; and Salakhutdinov, R. 2014. Dropout: a simple way to prevent neural networks from overtting. *Journal of machine learning research 15(1):1929 1958.*