CS230

Recreating AlphaZero Chess Engine

Mark Chang **Ting Liang**

Stanford University
CS230 - Deep Learning
{mrkchang,tngliang}@stanford.edu

Abstract

We proposed a project to recreate a chess playing engine based on Google Deepmind's AlphaZero paper. We used deep supervised learning to learn policy and value functions from 1 million current TCEC champion engine Stockfish self-play games. We also implemented Monte-Carlos Tree Search to evaluate chess positions at runtime to ensure adequate validation of the generated chess moves using trained policy and value network. Finally we ran chess tournaments between different checkpoints of our engine and against stockfish in simulated chess tournaments.

1 Introduction

Board strategy games, such as chess, have been a platform for humans and machines alike to study and measure intelligence in a well defined manner. Board games tends to have simple and well defined rules that are easy to understand, and enjoyable to play or watch. For decades, the Artificial Intelligence (AI) community has gravitated towards these games as benchmarks for the study of intelligence. In 2018, Deepmind paper on AlphaZero that simultaneously learned to master the game of Chess, Go and Shogi by complete self-play reinforcement learning with no human knowledge input has been hailed as a significant break-through.

The goal of our project is to reproduce the AlphaZero chess engine and perform an engine match with the open source Stockfish engine, which is the current TCEC champion chess engine. By implementing this project we hope to learn and enhance our skills in constructing deep reinforcement Artificial Neural Network (ANN) architectures, Monte Carlo Tree Search (MCTS), Policy Gradient Updates, as well as tuning and refining deep ANNs in the supervised setting.

2 Related works

For a long time Chess has been the "drosophila" of AI development. In 1997 IBM's deep blue first beat human world chess champion Kasparov. Since then Chess engines have adopted various versions of minimax search with alpha-beta pruning and Grandmaster handcrafted chess features to definitively achieve superhuman play. The 2017 TCEC champion engine is stockfish that is rated with elo rating 3232. By comparison the best human has only achieved elo rating of around 2800.

In Dec 2017 Google's Deepmind published AlphaZero paper that used the same ANN architecture to achieve superior performance in chess, go and. The paper followed their initial success with AlphaGo that achieved super-human performance famously against Korean go master Lee-seedol and then Chinese go champion Ke-jie, and is achieved entirely by training on self-play games with no human knowledge input.

Other related work in reinforcement learning includes OpenAI's Proximal Policy Optimization that seeks to avoid performance collapse due to bad policy update by clipping each policy gradient update to the KL divergence between old policy and new policy network. This is necessary because bad policy network in reinforcement learning can have a significant adverse impact on subsequent learning due to generation of bad data.

3 Approach

Initially our plan was to recreate the chess engine exactly according to Google Deepmind's AlphaZero paper as is. However, lack of computation resources quickly led us to switch our approach. Deepmind's paper used 5000 1st generation TPUs in parallel to generate self play games from the latest trained ANN and feeds these self-play games into 16 learners to update the ANNs. Each TPU is equivalent to one fastest GPU on the market. However our project computation budget consists of entirely our own personal computers and one p3xlarge AWS instance with 1 GPU. We found that generating a self-play game using a pair of ANNs alone required 13 minutess of GPU time with thousands of games required to begin training. And since the initial batch of games were observed to also be of bad quality that consists entirely of random plays, it became obvious the project cannot succeed using the same data generation methodology as Deepmind did in the allotted time frame.

Our new methodology would use the same ANN architecture, but instead of learning from self-play games, a single learner would study Stockfish generated self-play games. Since Stockfish is entirely CPU based and CPUs are much cheaper to access than GPU, we were able to generate 1,000,000 Stockfish self-play games in 2 days to kick off the training process. At evaluation time our engine would still use ANN with its own Monte-Carlos Tree Search (MCTS) algorithm to try beat Stockfish at its own game.

4 Datasets

Our dataset consists of 1,000,000 Stockfish v Stockfish self-play games using 4 desktop computers generated over 2 days. All games are saved in text format using Python Chess's PGN writer, and organized in 10,000 game batches per file (100 files totals 1GB in size). We additionally generated 10,000 games for dev set and 10,000 games for test set using the same methodology.

Preprocessing We ran into a challenge with Python Chess package used to handle PGN and chess game rules leaked memory and halted training after only 3-4 steps due to Memory Exhaustion. This happened even after we eliminated our ANN. To work around this issue and load training data faster, we preprocessed all 1,000,000 PGN games into .h5 files using h5py package. This consists of loading 1,000 PGN games each time, decoding every state and action history to numpy arrays, and combining them with results and legal actions to store everything in .h5 files as datasets under the group named after the game batch count. Preprocessing took an additional 48 hours for the 1,000,000 generated self-play games. To avoid the memory issue we shut down the running process after each 1,000 games and restart processing from a new process using a command script.

Chess Board State, Action, and Result Representation We implemented the board state and action space representation outlined in AlphaZero paper.

Each chess board is represented by a state matrix of 119x8x8 volume, where 14 planes are used to encode each of the 6 different chess pieces (King, Queen, Rook, Knight, Bishop, and Pawn) on separate planes from both sides, plus 2 planes of repetition counts for both players. The board is repeated 8 times to represent last 8 half moves. The final 7 planes encodes global aspects of the game such as player's color, king and queen side castling rights for both players, total move count and no progress move count (See Figure 1).

Every chess action is represented by an action matrix of 73x8x8 volume of which 56 planes are queen style moves from a given square in each of 8 directions for between 1 to 7 steps, 8 planes for knight style moves, and 9 planes for 3 under-promotion moves to rook, bishop and knight respectively.

Finally, every chess game result is annotated with a -1 (loss), 0 (draw), or 1 (win).

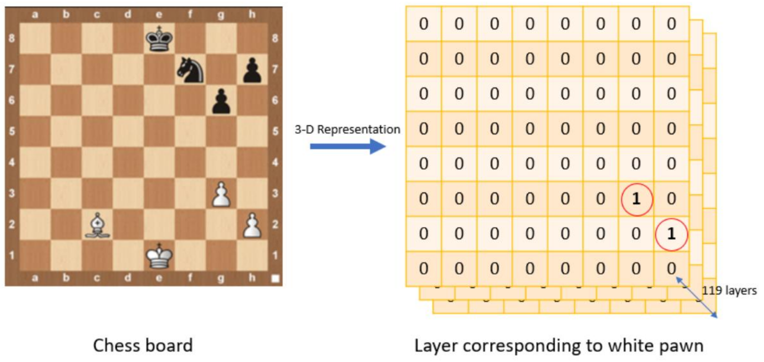


Figure 1: 3-D State Representation

5 Model Architecture

We implemented AlphaZero's neural network architecture in both tensorflow and pytorch. It consists of (None, 119, 8, 8) input state fed into 19 resnet blocks each consisting of 2 batch-normalized convolutional layers with skip connections, using 256 3x3 filters and stride 1. The output of the resnet is split into a policy head and value head separately to facilitate multi-learning. The policy head consists of a batch-normalized convolutional layer with 256 1x1 filters and stride 1 followed by another batch-normalized convolutional with 73 1x1 filters and stride 1. The policy head final layer logits are not activated. Instead they are fed into the loss function as logits directly. The value head consists of a batch-normalized convolutional layer with a single 1x1 filters, followed by 1 dense fully-connected layer and tanh activation. All convolutional layers have padding to maintain dimensions.

Our loss function is also the same as one used by AlphaZero.

$$loss = (v - \hat{v})^2 - \pi^T logp + c\|\theta\|^2$$

where,
v- : The mean squared error between target value v (actual game outcome +1/0/-1 (tanh activation)) and predicted values
π^Tlogp : The softmax_cross_entropy_with_logits_v2 loss between target policy (one hot Stockfish action pick) and our ANN policy head logits. tf.stop_gradient was applied to the target policy label to stop gradient from flowing into the label.
c$\|\theta\|^2$: The l2 regularization for all model trainable parameters θ with weight decay parameter c

5.1 Modification to Model Implementation

When we began training our model, we soon discovered it was very difficult to train as implemented. After some investigation we determined to root cause to be oftmax_cross_entropy_with_logits_v2 being applied to all 672 (73x8x8) classes of labels. This was too large for the given datasets to produce good results. A typical chess position would only produce 20 valid moves on average, so 4672 includes mostly illegal moves which doesn't need to be learned from the environment.

As a result, we made the following modification to the loss function calculation to speed training:

1. We added a legal action Boolean mask of size (73, 8, 8) that specifies True for legal actions under the current input state, and False otherwise.
2. All policy logits are then masked into a single vector of legal policy logits of dynamic size (using tf.boolean_mask), usually around 20 legal moves (classes) for chess.
3. Loss functions only adds unmasked policy logits, the rest of masked policy logits are discarded for training.

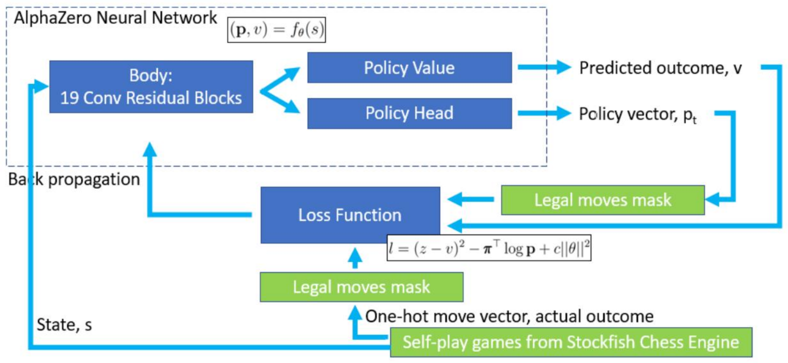

AlphaZero Neural Network
$(\mathbf{p}, v) = f_\theta(s)$
Body: 19 Conv Residual Blocks
Policy Value
Policy Head
Predicted outcome, v
Policy vector, p_t
Back propagation
Loss Function
Legal moves mask
$l = (z - v)^2 - \boldsymbol{\pi}^\top \log \mathbf{p} + c\|\theta\|^2$
Legal moves mask
One-hot move vector, actual outcome
State, s
Self-play games from Stockfish Chess Engine

Figure 2: Modifications to AlphaZero Training (Mods in green text)

6 Training, Hyperparameter Tuning and Monitoring

First we load the dev set at the beginning of training (1000 games in one .h5 file). Then we load the train set 1 batch at a time (1,000 game in one .h5 file).

Then at train time we would evaluate [value, policy_logits, loss, grad_norm] tensors separately on sampled chess state and moves from both the current train set and dev set. The results of the evaluation is recorded into tensorboard summary, and also logged to console.

Batch size The batch size for number of training chess moves to sample were reduced from AlphaZero paper's specification of 4096, first according to GPU memory availability (768 for desktop and 64 for laptops). It was further reduced to 128 for desktop after observing better training speed. Finally it was observed batch size of 64 on laptop actually obtained smaller loss values after only 1600 steps of training.

We introduced a hyperparameter num_steps_per_games to determine how long we resample chess moves from the same batch of 1,000 games loaded in memory before moving onto the next batch of 1,000 games. Initially we tried 2,500, then 1,000, and finally settled on 100. Since there are 140 moves on average per game, each 100 games contains 14,000 state action pairs, which is sufficient to capture most of the data point available from each batch of games. The dev set loss has dropped in tandem with train loss, and is slightly higher typically by 1%.

Model resnet block count We experimented with the number of resnet block count in the model from 19 to 1 and 5, but observed much higher loss floor using the original 19 blocks. Since the 19-block model still fits in laptop GPU's memory with batch size of 64, and also performs better, so we left this parameter alone.

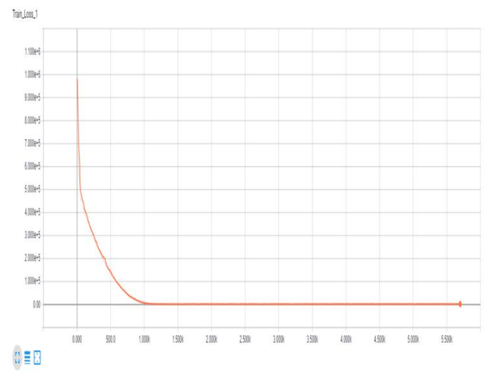
Weight decay (L2 regularization) We tried various weight decay parameters and found the original AlphaZero value of 0.0001 was too low and caused the loss error to explode at the start of training. This at times resulted in numeric overflow in the softmax exponent calculation. After some experimentation we found that weight decay that scales linearly with batch size worked best. 700 for batch size of 768, 100 for batch size of 128, and 60 for batch size of 64 worked best.

Adam Optimizer and gradient clipping To further mitigate exploding gradients, we implemented gradient clipping by retrieving gradients_and_vars from AdamOptimizer.compute_gradients, and clip the gradients according to tf.clip_by_global_norm with hyperparameter clip_val. This is a simpler version of modification that was mentioned in the PPO paper, although that paper used KL divergence between old and new policy network. We found the simple constant clipping was sufficient to stablize the policy network with properly chosen weight decay and clip value.

Finally, we changed the momentum optimizer used in AlphaZero paper to AdamOptimizer. This eliminated the need for both momentum and learning rate schedule hyper-parameter from the original AlphaZero paper.

7 Results

With batch size 128 and weight decay 100, the training loss decreases from 1,000,000 quickly to a floor (oscillating between 1000 and 2500) after only 1200 steps (each weight updates using a minibatch is considered a step)


(a) train set loss

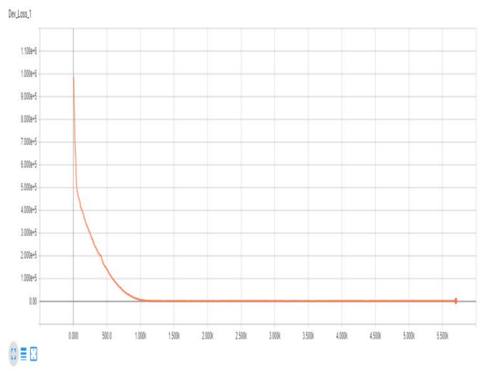

(b) dev set loss

Figure 3: Observed losses over train steps

The dev loss was calculated at the same time as train loss but using 1000 games not used during training.

The final test loss was measured using 1000 held-out test games after 5699 steps with a loss of 4092.32.

With all agents start at elo rating of 1000, the final tournament was played among checkpoint 0, 689, and 5699. The final elo scores for 689 and 5699 achieved against checkpoint 0 are respectively 1030 and 1052 in simulated tournament plays between checkpoints.

We also manually inspected the first few moves generated by the trained agent at checkpoint 5699 compared with the initial checkpoint 0 and found it has learned from stockfish the more common e2e4 and d2d4 opening move rather than edge pawn move a2a4. This behavior makes sense since initial board state is the most common state it has seen in the training set.

8 Conclusion

Our supervised deep learning method using stockfish self-play games can only captured one-hot encoded stockfish best moves in the target policy. This differs from AlphaZero's design to learn the visit counts distribution and extract best move from it. So our ANN was only able to generalize marginally with 50 point elo gain, which is significantly lower than 2200 points required to beat Stockfish. We therefore confirm AlphaZero paper's original premise that self-play reinforcement learning from scratch is the only way to achieve superior performance in a game that require high level of strategic thinking.

9 Future Works

To properly train AlphaZero, more computation power need to acquired, especially on the self-play game generation side which is currently limited to 16mins GPU time per game. We also need to further explore how to speed up MCTS algorithm that is limiting the speed of self-play game generation due to ever changing visit counts creating dependency between each simulated game to the next and forcing all simulation games to be played in serial. Finally we need to persist visit counts in self-play games along with network weights and use importance sampling on learner to update the ANN weights asynchronously but correctly.

10 Contributions

Both members contributed equally throughout the project. Ting set up most of the chess playing infrastructure with Python Chess and UCI communications, state and action space encoding, completed AlphaZero self play, ReplayBuffer, Shared Storage, data preprocessing and training, modification of models using tensorflow, and obtaining final tournament results. Mark setup the same model in AWS using pytorch (In AWS cloud VM we were only able to get GPU to work with pytorch in Windows, tensorflow models were run on local desktop with a GE-Force 1080Ti GPU), and used the equivalent model to perform training and analysis in the cloud. Both members contributed to the poster and final report.

References

[1] David, Omid E, Nathan S Netanyahu, and Lior Wolf (2016). "DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess". *International Conference on Artificial Neural Networks. Springer, pp. 88–96.*

[2] Linscott, Gary. "LCZero." *lczero.org/.*

[3] Fiekas, N. (2014). Chess: A pure Python chess library¶. Retrieved from https://python-chess.readthedocs.io/en/latest/

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. *tensorflow.org.*

[5] Paszke, Adam and Gross, Sam and Chintala, Soumith and Chanan, Gregory and Yang, Edward and DeVito, Zachary and Lin, Zeming and Desmaison, Alban and Antiga, Luca and Lerer, Adam., et al. (2017). Automatic differentiation in PyTorch. *NIPS 2017 Workshop Submission.*

[6] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature, 529(7587):484–489.*

[7] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2017a). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. arXiv preprint *arXiv:1712.01815.*

[8] Yang, Daylen. "Stockfish Chess." Stockfish, *stockfishchess.org/.*