Creating a Chess AI Using Deep Learning

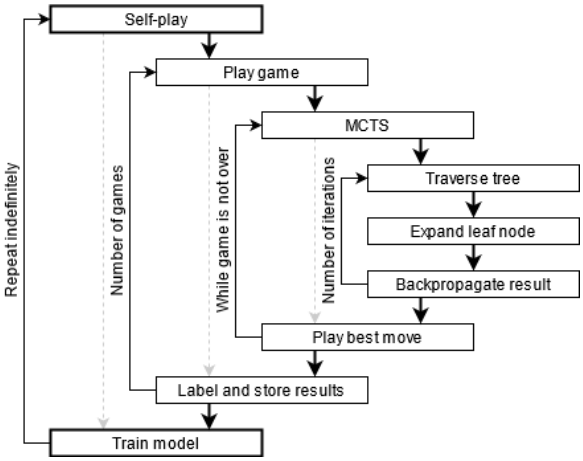
Introduction

The year 1997 marked a turning point in chess history, as a reigning world chess champion was defeated by a computer for the first time. Since then, chess engines have only continued to grow stronger, completely surpassing humans. The strongest engines were based on domain-specific adaptations and required handcrafted evaluation functions that were refined by human expers over several decades. However, that all changed in 2017, when AlphaZero was introduced. Given no domain knowledge except the game rules, it achieved a superhuman level of play in under 24 hours. More specifically, it outperformed Stockfish 8 in just 4 hours of training. AlphaZero's incredible achievement was the main inspiration for this project.

The initial goal of this project was to create a bot capable of playing chess using deep learning. The bot would initially be given only the rules of chess. It would then use reinforcement learning (playing itself) to master the game. To achieve this, I decided to make my own implementation of AlphaZero. Papers describing the algorithms of AlphaGo Zero [1] and its successor AlphaZero [2, 3] were used as the main reference.

Overview

The program was implemented in Python and PyTorch. It was designed to run on a single GPU. A high-level diagram of the AlphaZero algorithm used is shown on the right.



First, the program plays several games with itself and stores individual positions. The outcome of each game is then used to label the data. Next, the neural network model is trained on the self-play data. This cycle is repeated indefinitely.

When playing games, the program performs a Monte Carlo tree search (MCTS) on each turn in order to find the move with the highest probability.

Methods

Reinforcement learning

The AlphaZero model takes the current board state s as input and outputs a policy p, representing probabilities for each move, and a value v, representing the predicted outcome. During self-play, it uses a type of policy improvement and evaluation scheme. On each move, the neural network policy is used to execute iterations of MCTS. This yields a much stronger policy π, which is used to play the next move. When the game ends, the outcome z is obtained. For each position from the game, a training example (s_i, π_i, z) is stored. During training, the model aims to bring its predicted policy closer to the improved policy $p \to \pi$, and its predicted value closer to the game outcome $v \to z$. Thus, the following loss function is used:

$$l = (z - v)^2 - \pi^{\mathrm{T}} \log p$$

Monte Carlo tree search (MCTS)

MCTS works by building a tree of game states, which are connected by actions (chess moves). It starts off by creating a root node, which represents the initial search state. During each MCTS iteration, the tree is traversed until a leaf node is found. The leaf node is then expanded and its (estimated) value is propagated back to the root.

In order to expand a leaf node, the neural network model is first evaluated using the node's state s_L, yielding a predicted policy p and value v. For each legal move a, a child node is created with an assigned prior probability $P(s_L, a) = p(a)$, a visit count $N(s_L, a) = 0$, a mean-action value $Q(s_L, a) = 0$ and a total-action value $W(s_L, a) = 0$. The visit counts and values of previously visited nodes are then updated in a backwards pass for each step $t \leq L$: $N(s_t) \mathrel{+}= 1$, $W(s_t) \mathrel{+}= v$ and $Q(s_t) = \frac{W(s_t)}{N(s_t)}$.

When traversing the tree, action a_t at time is selected by maximizing its UCB (upper confidence bound) score, with $c_{init} = 19652$ and $c_{init} = 1.25$:

$$a_t = \text{argmax}_a\big(Q(s_t, a) + U(s_t, a)\big)$$

$$U(s, a) = C(s)P(s, a)\frac{\sqrt{N(s)}}{1 + N(s, a)}$$

$$C(s) = \log\left(\frac{1 + N(s) + c_{base}}{c_{init}}\right)$$

After the final iteration, the improved policy π is computed proportionally to the visit count of each child, $\pi_a = \frac{N(s_0,a)^{1/\tau}}{N(s_0)}$, where τ is a temperature parameter.

Input and output encoding

The input to the model is a $22 \times 8 \times 8$ image stack representing a chess board state. The 22 planes represent various features, while the 8×8 "images" are used to one-hot encode each feature's representation (except repetition and no-progress count planes, which are constant-valued). The input is always oriented to the player whose turn it is (P1).

Feature	Planes
P1 piece	6
P2 piece	6
Repetitions	2
P1 castling	2
P2 castling	2
En passant	1
No-progress count	1

The model outputs are a value scalar v and a 4672 dimensional policy vector p representing action probabilities. Each action consists of selecting the piece and choosing how to move it. Selecting the piece has $8 * 8$ possibilities, corresponding to the board size. Moving the piece is then split into 3 parts – queen like-moves, knight moves and underpromotions, encompassing 73 possibilies. This means each move can be uniquely represented by a $73 * 8 * 8 = 4672$ dimensional vector

Feature	Planes
Queen moves	56
Knight moves	8
Underpromotions	9

Model architecture

The model consists of a convolutional block followed by 19 residual blocks, and an output block. Unless otherwise specified, each convolutional layer applies 256 filters of size 3x3 with stride 1.

The convolutional (input) block consists of: [Conv → Batch norm →ReLU].

A residual block consists of: [Conv → Batch norm → ReLU → Conv → Batch norm →ReLU] along with a skip-connection to the second ReLU.

The output block consists of a policy and value head.

- The policy head consists of: [Conv → Batch norm → ReLU → Conv (73 filters) → Softmax].
- The value head consists of: [Conv (1 filter) → Batch norm → Fully connected(64×256) → ReLU → Fully connected(256×1) → Tanh].

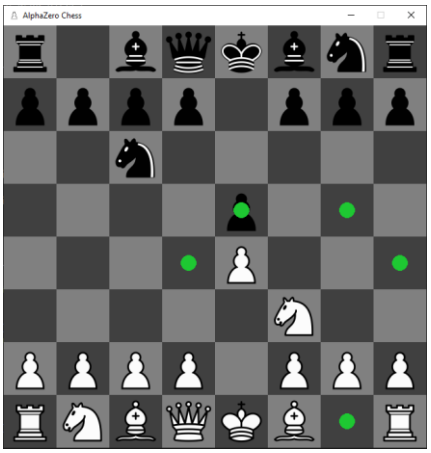
Training

The model was trained using two configurations:

- Reinforcement learning:
  - Epochs = 20, Batch size = 128
  - Optimizer = Adam, Learning rate = 0.1, Weight decay = 1e-6
- Supervised learning:
  - Epochs = 10, Batch size = 2048
  - Optimizer = SGD, Learning rate = 0.2, Momentum = 0.9

Results

In order to evaluate the results, I made a tool that allows the user to play against the trained model, as well as watch it play against itself. An example screenshot can be seen on the right.



Unfortunately, the model barely showed any improvement. When I played against it, I could easily beat it. It seems that a single GPU is not nearly enough computation power to handle the action space in chess. (As a comparison, the original AlphaZero used 5000 TPUs to generate self-play games and 64 TPUs to train the network in parallel).

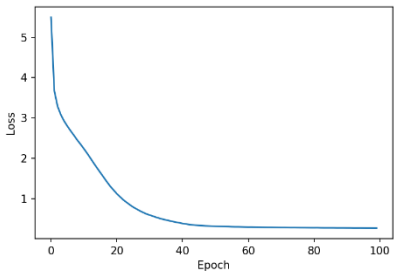
The self-taught model was trained for slightly over 50 hours. In that time, it managed to complete 25 steps of 15 self-play games. Each game had a maximum move limit of 300 and used 400 MCTS iterations per move.

The self-taught model was evaluated by playing against an untrained version of itself. The test was conducted by playing 100 games, using 10 MCTS iterations for each move. As white, the trained model won 26% of games, lost 16% and 58% ended in a draw. This shows that the trained model performs similarly as an untrained model.

```
Playing games: 100%|██████████| 100/100 [1:04:56<00:00, 38.97s/game, White = 26 | Black = 16 | Draw = 58]
White:  26.0%
Black:  16.0%
Draw:   58.0%
```

Supervised learning

Disappointed by the results of reinforcement learning, I decided to train the model on games played by humans. I used the FICS Games Database [5] to download all chess games played on their website in 2020 by players with an ELO rating of at least 2000. This dataset contained approximately 40,000 games and over 3,000,000 in-game positions.



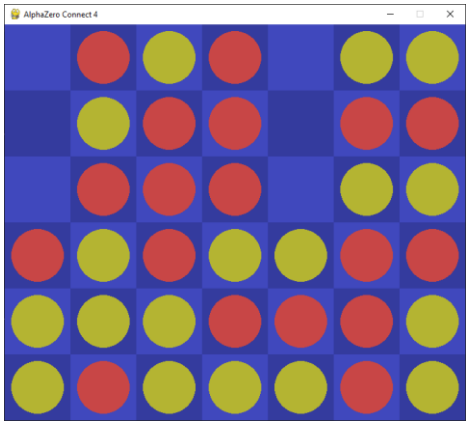
The network was trained for around 15 hours and completed 110 epochs of training. The image on the right shows average loss during training. To speed-up the learning, **the number of residual blocks in the model was reduced from 19 to 11**.

The supervised model was evaluated similarly as before – 100 games against an untrained version of itself, both using 10 MCTS iterations for each move. As white, the trained model won 48% of games, lost 3% and 49% ended in a draw. That is a significant improvement!

```
Playing games: 100%|██████████| 100/100 [34:34<00:00, 20.75s/game, White = 48 | Black = 3 | Draw = 49]
White:  48.0%
Black:  3.0%
Draw:   49.0%
```

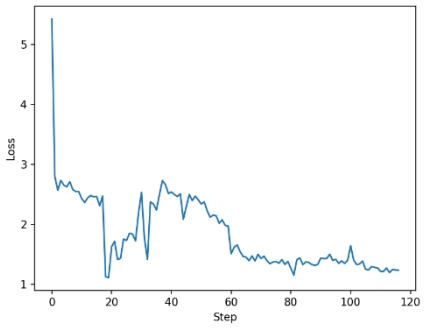
Connect Four

In order to test the reinforcement learning method in a game with a much smaller action space, I applied it on the game Connect Four. Once again, I developed a tool that lets the user play against the trained model, as well as watch the model play against itself.



This time, the trained model actually managed to beat me! However, the model is still not perfect. Connect Four is a solved game – meaning that yellow can always force a win. However, the model still loses against itself sometimes.

The network was trained for around 7 hours and completed 117 steps of 100 self-play games, using 25 MCTS iterations per move. The average training loss for each step is shown on the right.



The self-taught model was evaluated, once again, by playing 100 games against an untrained version of itself, both using 10 MCTS iterations for each move. As yellow (first to move), the trained model won 84% of the games and lost 16%. As red, the trained model won 67% and lost 33% of games.

```
Playing games: 100%|          | 100/100

Yellow: 84.0%
Red:    16.0%
Draw:   0.0%
```

```
Playing games: 100%|          | 100/100

Yellow: 33.0%
Red:    67.0%
Draw:   0.0%
```

References

[1] Mastering the game of Go without human knowledge
https://www.nature.com/articles/nature24270
[2] Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm
https://arxiv.org/pdf/1712.01815.pdf
[3] A general reinforcement learning algorithm thatmasters chess, shogi and Go through self-play
https://science.sciencemag.org/content/362/6419/1140/
[4] A Simple Alpha(Go) Zero Tutorial https://web.stanford.edu/~surag/posts/alphazero.html
[5] FICS Games Database - https://www.ficsgames.org/download.html