Published in Towards Data Science

Sayon Bhattacharjee Follow

Nov 13, 2018 · 9 min read · Listen

Save

Predicting Professional Players' Chess Moves with Deep Learning

Source: Pexels

So I'm bad at chess.

My dad taught me when I was young, but I guess he was one of those dads who always let their kid win. To compensate for this lack of skill in one of the world's

most popular games, I did what any data science lover would do: build an AI to beat the people I couldn't beat. Unfortunately, it isn't nearly as good as AlphaZero (or even the average player). But I wanted to see how a chess engine would do *without* reinforcement learning as well as learn how to deploy a deep learning model to the web.

Play against it here!

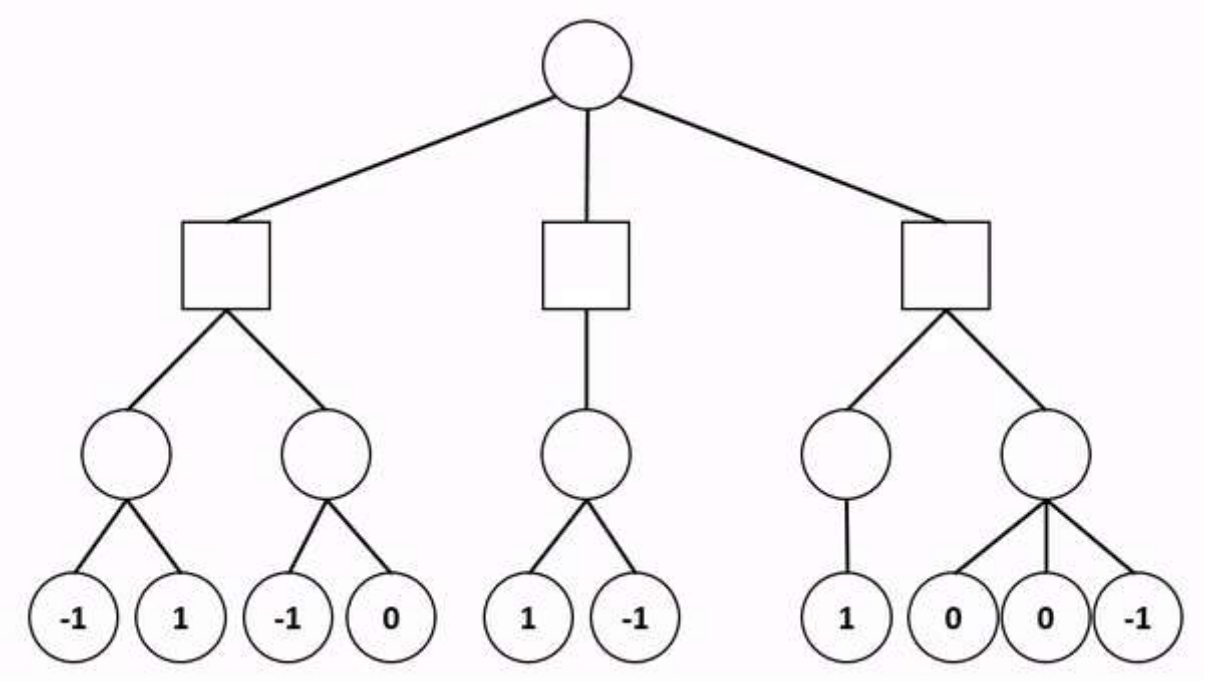
Getting the Data

FICS has a database of 300 million games, individual moves made, the results, and the rating of the players involved. I downloaded all the games in 2012 where at least one player was above 2000 ELO. This totalled to about 97000 games with 7.3 million moves made. The win distribution was: 43000 white wins, 40000 black wins, and 14000 draws.

Minimax

To understand how to make a deep learning chess AI, I had to first understand how a traditional chess AI was programmed. In comes minimax. Minimax is an abbreviation for "minimizing the maximum loss" and is a concept in game theory to decide how a zero-sum game should be played.

Minimax is normally used with two players where one player is the **maximizer** and the other player is the **minimizer**. The agent, or whoever is using this algorithm to win supposes that they are the maximizer, while the opponent is the minimizer. The algorithm also requires that there is a board evaluation function which is a numerical measure of who is winning. This number is between -∞ and ∞. The maximizer wants to maximize this value, while the minimizer wants to minimize this value. This means that when you, the maximizer, faces a crossroad between two moves, you will pick the one that gives you a higher evaluation, while the minimizer will do the opposite. The game is played assuming both players play optimally and no one makes any errors.



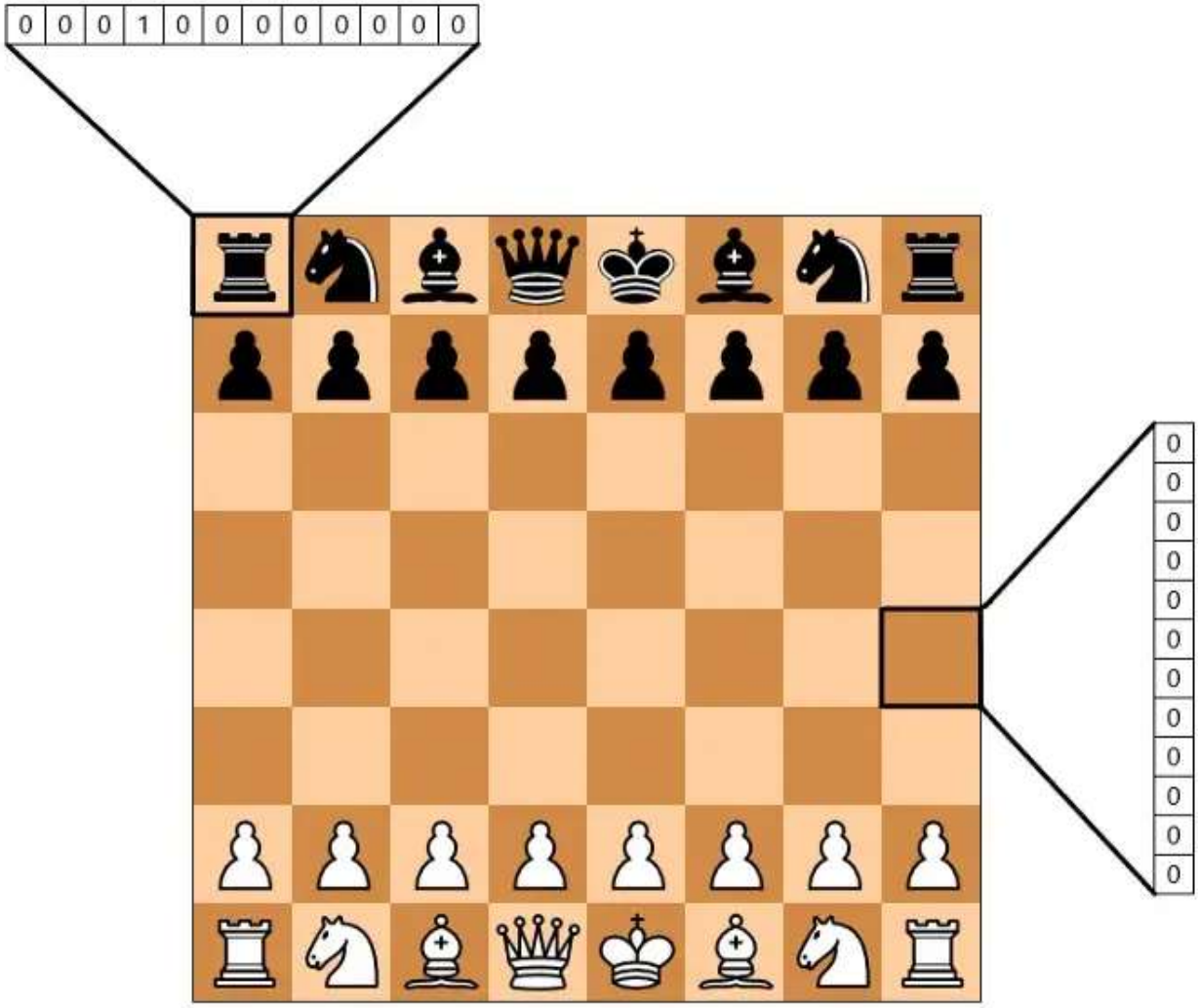
Source: GlobalSoftwareSupport

Take the above GIF for example. You, the maximizer (the circle), have three moves you can choose from (starting from the top). The move you should choose directly depends on the move your opponent (the squares) will choose *after* the move. But the move your opponent chooses directly depends on the move you choose after *that* move, and so on until the game is over. Playing until game over can use a lot of computational resources and time, so you choose a depth to go to, in the above case, 2. If the minimizer(the leftmost square) chooses the left move, you have -1 and 1 to choose from. You choose 1 because it will give you the highest score. If the minimizer chooses the right move, you choose 0 because that's higher. Now it's the minimizer's turn, and they choose 0 because that's lower. This play goes on until all the moves are covered or you run out of thinking time. For my chess engine, the engine assumes white is the maximizer, while black is the minimizer. If the engine is white, the algorithm decides which branch will give the highest minimum score, assuming the human chooses the lowest score every time it's their move and vice versa. For better performance, this algorithm can also be combined with another algorithm: alpha-beta pruning. Alpha-beta pruning applies a cutoff system to decide whether it should search down a branch or not.

The Deep Learning Architecture

My research began with Erik Bernhardsson's great post on deep learning for chess. He goes through how he took the traditional method of making an AI play chess and transformed it to use a neural network as its engine.

The first step is to convert the chess board into numerical form for the input layer. I borrowed Erik Bernhardsson's encoding strategy where the board is one hot encoded with the piece that is in each square. This totals to a 768 element array (8 x 8 x 12 as there are 12 pieces).



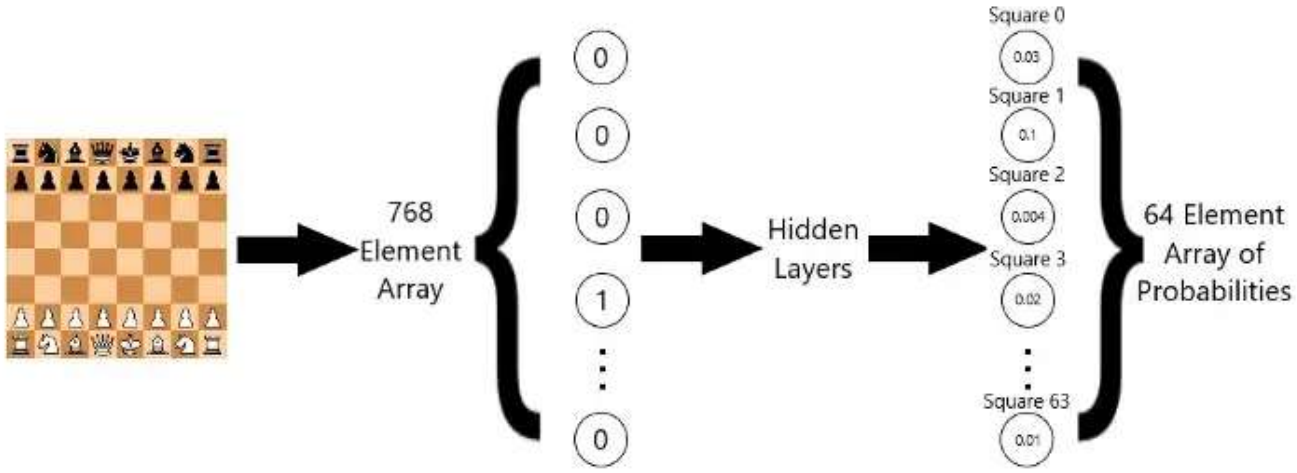
Bernhardsson chose to make the output layer a 1 for a white win, -1 for a black win, and a 0 for a draw. He assumed every board position in a game was related to the result. Every individual position was trained as "favours black" if black won, and "favours white" if white won. This allowed the network to return a value between -1 and 1, which would tell you whether the position was more likely to result in a white win, or a black win.

I wanted to approach this problem with a slightly different evaluation function. Would the network be able to see not whether white or black was winning, but be able to see which move played would result in a win? At first, I tried making the 768-element board representation into the output where one position would be the input and the very next position would be the output. Of course, this didn't work

because this turned it into a multi-classification problem. This allowed for too much error for the engine to properly choose legal moves, as all 768 elements in the output layer could be 1 or 0. So I consulted the Stanford paper Predicting Moves in Chess using Convolutional Neural Networks by Barak Oshri and Nishith Khandwala to see how they tackled the issue. They trained 7 neural networks of which 1 network was the piece selector network. This network decided which square was most likely to be moved from. The other six networks were specialized to each piece type and would decide where a specific piece should be moved to. If the piece selector picked a square with a pawn, only the pawn neural network would respond with the square most likely to move to.

I borrowed from both their ideas to make two convolutional neural networks. The first, the *moved from* network, would be trained to take the 768 element array representation and output which square (between square 0, and square 63) the pro moved from. The second network: the *moved to* network, would do the same, except the output layer would be where the pro moved to. I didn't take into account who won, as I assumed that all moves in the training data would be relatively optimal, regardless of the final result.

The architecture I picked was two 128-convolutional layers with 2x2 filters followed by two 1024-neuron fully connected layers. I didn't apply any pooling because pooling provides location invariance. A cat on the top left of an image is just as much of a cat as if the cat was on the bottom right of an image. However, for chess, a king pawn's value is quite different from a rook pawn. My activation function for the hidden layers was RELU while I applied softmax to the final layer so I would essentially get a probability distribution where the sum of the probabilities of all squares added up to 100%.



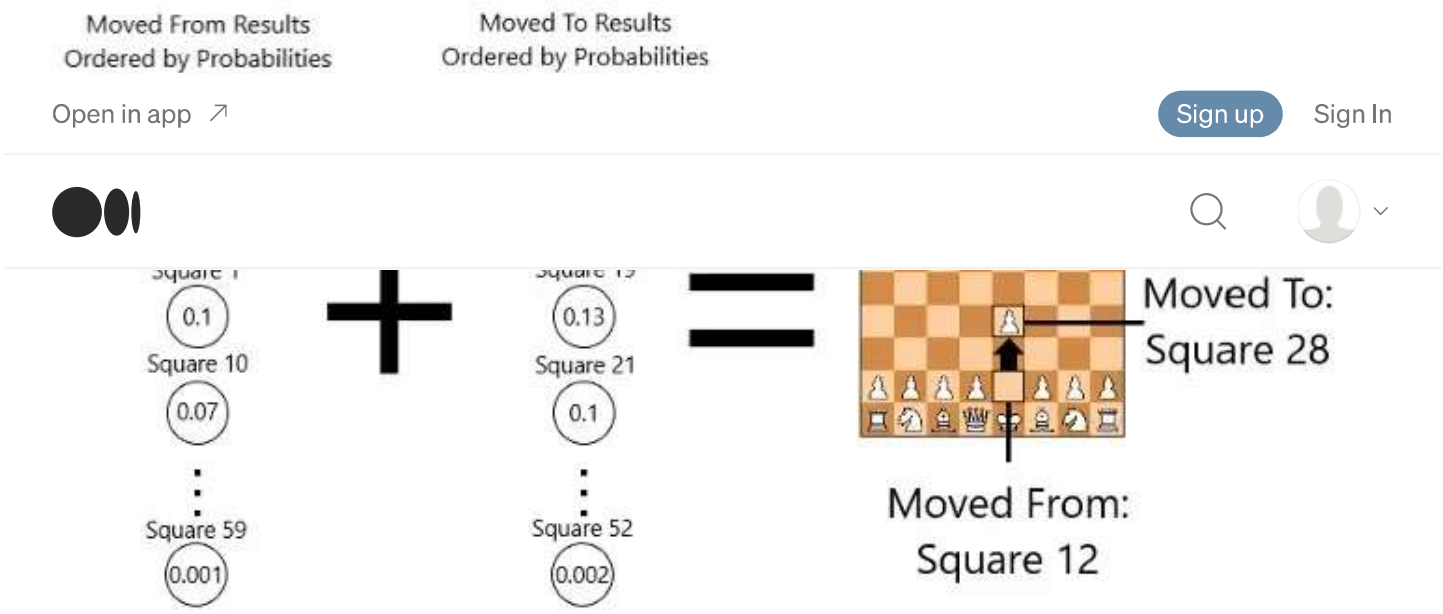
My training data was 6 million positions for the training set with the remaining 1.3 million positions for the validation set. By the end of training, I received 34.8% validation accuracy for the *move from* network, and 27.7% validation accuracy for the *move to* network. This doesn't mean that 70% of the time it didn't learn legal moves, it just means the AI didn't make the same moves as the pro players in the validation data. For comparison, Oshri and Khandwala's networks got an average validation accuracy of 37%.

Combining Deep Learning with Minimax

Because this is now a classification problem where the output can be one of 64 classes, this leaves a lot of room for error. A caveat with the training data (which was from the games of high-level players), is that good players rarely play up to checkmate. They know when they've lost and usually have no need to follow the entire game through. This lack of balanced data made the network heavily confused near the end game. It would choose the rook to move from, and try to move it diagonally. The network would even try to command the opposition's pieces if it was losing (cheeky!).

To tackle this problem, I ordered the outputs by probability. I then used the library python-chess to get a list of all the legal moves for a given position and picked the legal move with the highest resultant probability. Lastly, I applied a prediction score equation with a penalty for choosing less probable moves: 400-(sum of indices of moves picked). The further down the list the legal move is, the lower it's prediction score. For example, if the first index (index 0) of the *move from* network combined

with the first index of the *move to* network is legal, then the prediction score is 400-(0+0) which is the highest possible score: 400.



I picked 400 as the max prediction score after playing around with the number when combined with the material score. The material score is a number that would tell if the move made would capture a piece. Depending on the piece captured, the overall score of the move would get a boost. The material values I picked are as follows:

Pawn: 10, Knight: 500, Bishop: 500, Rook: 900, Queen: 5000, king: 50000.

This especially helped with end-game. In situations where the checkmating move would be the second most probable legal move and would have a lower prediction score, the material value of the king would outweigh it. The pawn has such a low score because the network thinks well enough early-game that it will take pawns if it is the strategic move.

I then combined these scores to return an evaluation of the board given any potential move. I fed this through a minimax algorithm of depth 3 (with alpha-beta pruning) and got a working chess engine that checkmated!

Deploying Using Flask and Heroku

I used Bluefever Software's guide on Youtube showing how to make a javascript chess UI and routed my engine through it by making AJAX requests to a flask server. I used Heroku to deploy the python script to the web and connected it to my custom domain: Sayonb.com.

Conclusion

Even though the engine doesn't perform as well as I hoped, I learned a lot about the foundations of AI, deploying a machine learning model to the web, and why AlphaZero doesn't use solely convolutional networks to play!

Possible improvements could be made by:

- Combining the *moved from* and *moved to* networks in a time series by using a bigram model LSTM. This might help with making the *moved from* and *moved to* decisions together, as each are currently approached independently.
- Improving the material evaluation by adding in the position of the material captured (capturing a pawn in the centre of the board is more advantageous than capturing it when it is on the side)
- Switching between using the neural network prediction score and material score instead of using both at every node. This could allow for a higher minimax search depth.
- Accounting for edge cases, such as: reducing the likelihood of isolating one's own pawns, increasing the likelihood of placing a knight near the centre of the board

Look at the code, or train a new network yourself with your own training data at the GitHub repo!

If you have any criticisms or concerns, you can reach me on LinkedIn, or my personal website.

Artificial Intelligence | Machine Learning | Deep Learning | Chess | Towards Data Science

---