Machine Learning Chess

Peter Flockhart, Henry Johnson
Department of Computer Science
Boston University
Boston, MA 02215
pflock,johnsoh@bu.edu

August 11, 2017

Abstract

The paper explains the methods taken to develop a machine learning algorithm to play chess using a neural network. The complexity of chess requires training the computer on a large database of past chess games to effectively evaluate chess positions in order to play high level chess. The computer was given as little information about the game of chess as possible so that it may learn the rules of chess by observation.

1 Introduction

As machine learning advances rapidly in the twenty-first century, an obvious application is to gaming. For centuries games have entertained and challenged their players. The inherent nature of humans is to pursue greatness, to be set higher standards and better records. With the introduction of intelligent machines, humans have introduced a new challenge and adversary amongst which they feel the need to "compete" against, the computer. The competitiveness of human nature has led to amazing discoveries of technologies and new methodologies of thinking and learning to gain the upper hand on the opponent. When confronted with a machine, something humans view as more intelligent, more logical, the question "Can I beat it?" arises. No game is more rich in history or so deeply thought of than chess.

Chess is a highly mathematical game, requiring future-thinking, instantaneous analytical skills, and creativity in order to beat your opponent. With its 32 pieces, 64 squares, 7 unique movement patterns, and various rules, it is no surprise that the Shannon number, 10^{120}, is the conservative lower bound of the game-tree complexity of chess. With one-hundred trillion neural pathways, the human brain sees unimpressive by comparison. It is recreating these pathways, this neural network, to create an intelligent, chess-playing machine capable of beating even the best players that is the subject of this paper.

2 Neural Network

In order to create a model to quickly evaluate a chess position with a high accuracy/probability of giving the Best Move (defined in section 3.3), a neural network must be used. The problem is too complex for any computer program to run, and requires machine learning so that the model may learn the rules of chess and how to play, without being explicitly told.

2.1 Datasets

A comparatively small amount of data was used to train the model. Time constraints limited the gathering/generating, parsing, and formatting of data. Instead of the ideal one- to ten-million samples, the model was trained around two-hundred-thousand samples. The data is sampled from pgn files containing thousands of games formatted in the EPD style.

2.2 Technology

Using Keras on top of TensorFlow in Python allowed for easy coding, implementation, and construction of a Neural Network. This helped abstract out most of the difficult work and math that goes into creating a machine learning algorithm. It is a wonderful tool for the simplification of complex problems and allowed for a decrease in development time, allowing for time for testing, training, and design, than if it were all hand-coded without Keras.

2.3 Model 1: Piece Count

The first model is a baseline; it uses the most basic of chess inputs and a relatively small neural network.

2.3.1 Structure

Model 1 has an input layer, 4 hidden layers, an activation layer, a dropout layer, and an output layer. All layers use a "tanh" activation function: $tanh(x) = \frac{2}{1+e^{-2x}} - 1$. Model 1 uses stochastic gradient descent to sweep through the training set, performing the update for each training example. We pass over the training set until the algorithm converges. The equation for stochastic gradient descent is $w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E_n$. In the model, τ is the epoch number and η is the learning rate. The learning rate is .1. A learning rate decay of .002 is used, along with the dropout layer, using a dropout rate of .2, to decrease the probability of overfitting the data.

2.3.2 Input

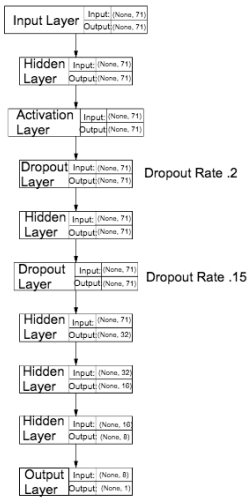
The input for Model 1 is twelve nodes. Each node represents the number of each type of piece on the board, e.g. 1 black king, 2 white rooks, etc.

2.4 Model 2: Piece Position

This second model is much more complex in input and structure that the first.

2.4.1 Structure

Model 2 has an input layer, 5 hidden layers, an activation layer, 2 dropout layers, and an output layer. All layers use a "tanh" activation function in this model, too. Model 2 uses stochastic gradient descent as well, however in this model, τ is the epoch number and η is the learning rate. The learning rate is .01. A learning rate decay of .002 is used, along with the dropout layers, the first with a dropout rate of .2, the second with a dropout rate of .15, to decrease the probability of overfitting the data.



2.4.2 Input

The input for Model 2 is 71 nodes. The first 64 nodes represent each square of the board, with a value corresponding to the piece at that location. The last 5 nodes represent castling rights, check/checkmate, and the turn.

2.5 Output

Both models produce the same kind of output, the evaluation of the chess board. Due to the use of the tanh function, output scaling was required to fit the data between -1 and 1, as the tanh function is a rescaled version of the sigmoid function to fit in the range [-1, 1]. The corresponding number, that between -1 and 1 is the indication of the evaluated position's score. When the number is closer to -1 the chosen position is a good move for black. If the number is close to 1, the move is good for white. The Best Move is indicated by the Max/Min of the evaluated moves for the board.

3 Evaluation

For both models, the metrics of success were how quickly the model runs and how accurate the model is in outputting the Best Move, and how often the Best Move was in the Top Three results of the returned output.

3.1 Model 1: Piece Count

3.1.1 Metric: Time

This model ran in a time of 6 seconds, evaluating all 200,000 data samples in that time, resulting in one position every 3-millionths of a second. This yields very good accuracy per unit time for the model. It is much faster than a traditional chess engine.

3.1.2 Metric: Accuracy

This model yielded an accuracy of 7%, three-and-a-half times the accuracy of picking the best move at random, for how often the Best Move was picked to use. The accuracy of the model giving the Best Move in the Top Three Moves is 18%. These are good for such a basic model, but there is room for improvement. Adding extra layers to the model was not good, and resulted in a dropoff in accuracy.

3.2 Model 2: Piece Position

3.2.1 Metric: Time

This model had the same runtime as Model 1, however, as will be seen in a decrease of accuracy, it suffers a close-to-random accuracy per unit time, which is not good.

3.2.2 Metric: Accuracy

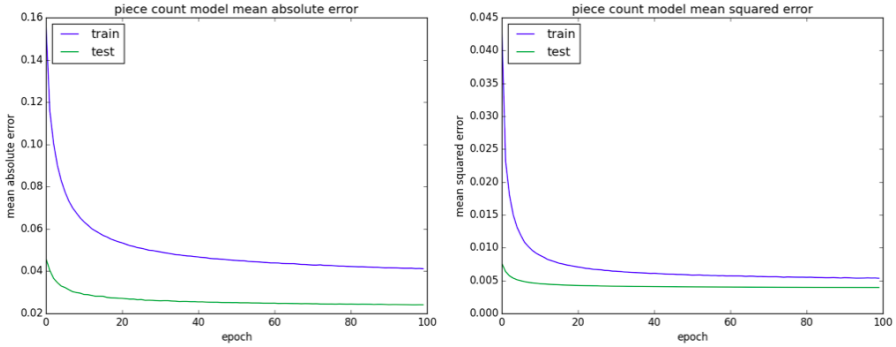
This model yielded an accuracy of 4.5%, only twice as much as picking the best move at random, for how often the Best Move was picked to use. The accuracy of the model giving the Best Move in the Top Three Moves is 10.66%. This was a surprise for such a complex model with many inputs that were believed to have the ability to increase the accuracy of the model compared to Model 1.

3.3 Notes on Evaluation

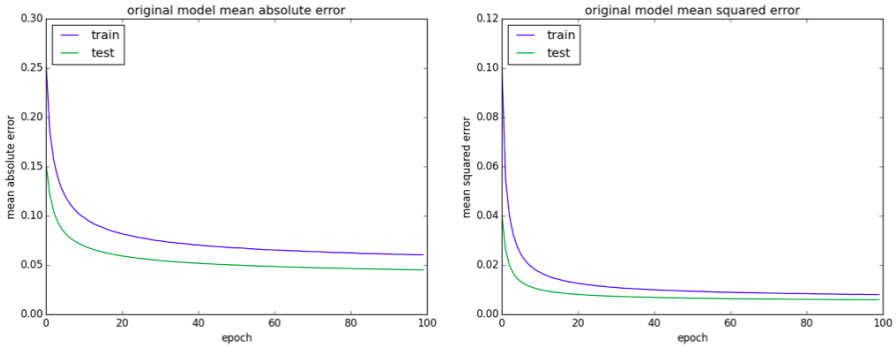
The error of the models is only relevant among many samples, not just one position because you need to know strength of a position relative to other positions. To define random, clarifying the accuracy by setting a random baseline, out of 50 possible chess moves per turn, picking one at random to play is 1/50, or 2%. For both models, the error graphs fit as would be expected with the 80/20 rule. While the decrease and stabilization of the mean squared error points to a good

learning rate and a predictable model, it does not necessarily correspond to a strong chess engine. Furthermore, it should be clarified that the percent of time the best move is picked out of 50 moves is the most helpful for assessing the model's understanding of chess positions.

Model 1 Graphs



Model 2 Graphs



4 Conclusion and Further Work

After reviewing the results of the model and its outputs as described in the above section, expectations of the project were both over- and under-whelming. The evaluation of the time metric was a success; with one position evaluated in three-millionths of a second, the model performed much quicker than a traditional chess engine, which evaluates one position on average every second. The evaluation of the accuracy metric left room for improvement, which will be outlined below. Model 1's accuracy for the Best Move being chosen, at 7% is acceptable, but can be improved. The Best Move in the Top Three accuracy for Model 1 at 18% is unacceptable, as are the corresponding Model 2 accuracy metrics.

4.1 Data Improvement

One method of improving the accuracy would be to increase the evaluation time of the data sets we test on to be > 1 second, for better position evaluation per game.

4.2 Accuracy Improvement

In addition to improving the evaluation time on our test data, training on a larger data set, possibly for a longer time (though it is not expected that this would make a significant difference as our models stopped improving and converged between 50 and 100 epochs of runtime), with reinforcement learning (feeding results of test back into the model to increase accuracy and develop a "smarter" machine; finer-tuned results and better understanding of chess theory/rules) would result in better accuracy of the model.

4.3 Further Testing and Implementation

Further testing would be to compare our outputs in a game played against a computer. This could give real time results about how well the model works and how it responds to live inputs, giving live outputs. If there is a significant increase in a win/loss ratio compared to without the model's optimization, then it would be a success.

Contributions

Below are the roles each member of the project played in developing the model and writing the report.

Peter Flockhart: Finding data and evaluation/generation, model coding and testing

Henry Johnson: Finding data, model testing, presentation and report creation/writing.