Turner Romey
CS-UY 4563: Intro to Machine Learning
Final Project
Project Members: Turner Romey

Pawn_Solo

Abstract:

The goal of the project is to create an AI that plays chess using machine learning. Machine learning has solved chess in the past using a combination of a deep neural network and a process called a "Monte-Carlo Tree Search." The goal of my project is not to solve chess, but rather create a chess bot that will read a board and react with what it thinks is the best move:

$$f(Board) = move$$

Normally, this decision is made using a process called Mini-max in which two players, taking alternating turns, descend a decision tree to determine which final state is better depending on an initial state. The strength of a board for a player is determined by an evaluation function:

$$f(Board) = \sum(MyPieces) - \sum(OpponentPieces)$$

This is important to understand because while a move may initially be beneficial for a player (maximizes f), the position could lead to a deficit or even loss the following turn. Because of this, most chess AI is written using minimax with a lot of hardcoded edge cases, overseen and approved by grandmasters. Machine Learning, specifically a neural network, can do this job of finding the edge cases, and the function shouldn't require a minimax algorithm. In this project, the end goal is a trained neural network using data that can, much like reading an MNIST image, read a board and return if a position is favorable for Black or for White.

Concept & Plan:

The machine learning tool which can "learn" to play chess based on board positions is the Convolutional Neural Network. Much like how a picture is represented, the input to the CNN will be a block with dimensions 5 x 8 x 8. This will be used to perform "convolutions" on as the "board" input is shifted to smaller sizes, ultimately finishing at a 1 x 128 output. This final output will be fed into a single "dense" layer which will return a single scalar output. This scalar output will read if the board position is better for the black (-1) or the white (1) player. This is the process for how to use the tool of a CNN to evaluate board positions.

When representing the chess board as a “picture,” that means we need channels. The 5 channels of the chess board are represented as the “states” a chess board has. These states include: pieces, positions, en passant, castling, and the player turn. States are read from the python Chess class based on current board. Python Chess has many methods and functions that allow the computer to receive a value to indicate a current state. It also allows for an easy parsing of PGN data. Which is essential for pre-processing.

PreProcessing and Data:

In order to train the tensor flow neural network we need a data set with an X and Y. Our data set would have to be a game of chess, and luckily there are tons of grandmaster games which can be easily accessed online in huge caches. The file format that these games are stored as is called PGN which is essentially a txt file holding the documentation of chess games. A single PGN text file holds thousands of games. Each game is given an Event, Site, Date, Round, Player White, Player Black, Result, ECO, and ELO in addition to the move order notation. Because all chess games start from a single starting position, we can recreate the entire game using Python Chess and the list of move orders. Additionally, by using “result” we can give each data set a winner which can help us in deciding what board representations are better for black or white. A game which black one would probably have better black board control. Our X data will be our 5 x 8 x 8 board representation and our Y data will be the “result” or who won. This is determined by ‘1/2-1/2’, ‘0-1’, or ‘1-0.’

In order to get this data, Python Chess will parse the txt file and generate a vector of moves of the type string in order to replay the game. Python Chess allows us to “push” a move to a current board state. For each move pushed, we can then take the 8 x 8 board that Python Chess returns and process it into our desired data block. This is then appended to the X array and the game’s result is appended to the Y array. These two arrays are then saved as a ‘.npz’ file which python can read and write raw data too.

It was originally assumed that all grandmaster game moves would be considered the optimal move. This is not the case, as the computer doesn’t understand the before and after of each move, only the position of the current pieces and state of the board in relation to who inevitably won. There is a loss of information here that the computer has trouble catching up with in terms of how to play. It was also considered to remove bad data like games that have large material deficits. However, this was never implemented like in the original proposal. It is simply universally assumed that at the end of each turn, a player is in a favorable position.

Model:

With the data pre-processed, we can now move on to the model design. The model, appropriately called "TensorChess" is broken down into 4 layer 'sections' labeled a, b, c, and d. Each section consists of 3 2D convolutional layers. Each Conv2D layer is given specifications for filters, kernel size, padding, and data format. Filters determine the dimensionality of the output space, or the number of output filters in the convolution. This means that in the first layer we are turning our board block into a series of 16 filters. The kernel size determines the height and width of each filter, so with the kernel size being equal to 3, we are using a filter to check a 3 x 3 area each time we perform a convolution. Padding is either "valid" or "same" and determines if the board has a border of blank spots or not. This allows our filter to get a more unbiased view of the edges of the board. Finally, data format tells the tensor flow Conv2D if the data is presented as channels first: 5 channels x 8 spaces x 8 spaces, or channels last: 8 spaces x 8 spaces x 5 channels. I had to change how I decided to represent the data because for some reason 'Channels_First' would lead to an "out of bounds" error around layer C1. By changing the pre-processing code to make channels last or 8 x 8 x 5, I was able to get the TensorChess to train on a small dataset.

The model was compiled before feeding it the X and Y, and in that compile I had to specify the loss, batch size, and optimizer. Because of the task, I lead to use mean-squared error, or MSE, as the loss function. MSE works best with gradient descent as GD is used to minimize the loss function and requires negative values. Because of this, I avoided loss functions that require an absolute value as GD needs to take a derivative to function. MSE was the easiest to visualize. For the batch size, I decided to choose 256. Batch size, to my understanding, seems to be an arbitrary decision with higher being more accurate and lower being more inaccurate. This is due to its effect on variance when averaging the data. My small laptop could only handle a moderate batch size of 256. For the optimizer, I chose to use ADAM as it is computationally cheapest. Finally, I used tanh() instead of sigmoid for my activation function. This was because while testing sigmoid, the network did not seem to learn at all. And when tanh was tested, I decreased the loss by 0.02.

This network, upon completion of training, would then be saved to the folder as a pre-trained model to be used to compute board values.

Problems and Solutions:

Many problems occurred with formatting and transformations of the data by tensorflow. These problems found bandaid solutions that I myself don't fully understand. One simple problem was the prevelance of "none" type data in my Tensor. Because the data was already in the form of a Tensor, I couldn't cast the Tensor as a single type. So I had to rep-process the data while casting it all as tf.int64 for a uniform dataset. The largest problem I ran into was that the loss never got below 0.6978. This is an extremely high loss and extremely inaccurate. It is probably due to the data set not being large enough, overfitting, and perhaps the way Im representing the chess board in pre-processing.

One problem with using a neural network is overfitting the data. A solution that other projects have used to some degree of success is to use what is called a "dropout" layer. I never attempted to use a drop out layer in a working model and I suspect that it would dramatically help reduce the loss. I would also like to try additional layer or fine-tune the conditions for each of the layers to minimize loss. One example is the output where it would've been a sigmoid from -1 to 1 where -1 is if black is favorable and 1 if white is favorable. However, I later chose tanh as it showed a slightly better loss.

Evaluation:

In the original project plan, I wished to use my Pawn_Solo to play against other players online. I would accomplish this by playing an online game on chess.com, and then having the Pawn_Solo bot give me a returning move for every move id give it from the player. In this way the AI would be playing against the player without ever interacting with the website. Id essentially be the middle man as I'd communicate the move between both applications. This test never happened as I believe the AI to be quite dumb. I also was unable to get it to properly train and it ended up being too late.

I do hope to finish this project and use it as a portfolio piece, where in which I'd test Pawn_Solo against the AI made by the people of the blogs and githubs I used as guides. I'd then love to look at the data, and maybe find some relationships within the data. I can do this by connecting both AI's to Python Chess and then bouncing the turn back and forth between them. I'd then record the games as their own PGN files along with other useful statistics such as turn number, piece deficit, or even an evaluation of each game by a more serious chess AI like Leela Chess. I think a report of the analysis of that data would be very interesting to write.

Ultimately this was a terrific learning experience for how to solve complex, rule-based, problems using an algorithm like a neural network, and a process like convolutions. I think representing game rules on the board as states and the representing states as channels is the correct way to approach this problem. I would like to consider perhaps trying more states of more specific types, perhaps differentiating the board state further between players. However, because the CNN seems to already be overfitting, perhaps more data is not the best idea. Maybe I should in the future look at reducing the data complexity in hopes of finding some deeper truths for how to represent the rules of chess.

References:

Data:

https://chessproblem.my-free-games.com/chess/games/Download-PGN.php

https://www.pgnmentor.com/files.html

http://www.kingbase-chess.net/

Papers:

https://pdfs.semanticscholar.org/28a9/fff7208256de548c273e96487d750137c31d.pdf

https://papers.nips.cc/paper/1007-learning-to-play-the-game-of-chess.pdf

https://www.cs.tau.ac.il/~wolf/papers/deepchess.pdf

Github Projects & Blogs:

https://github.com/LordDarkula/KnightSky

https://github.com/erikbern/deep-pink & https://erikbern.com/2014/11/29/deep-learning-for-chess.html

https://github.com/thomasahle/sunfish

https://github.com/undera/chess-engine-nn

https://github.com/Zeta36/chess-alpha-zero

https://machinelearnings.co/part-1-neural-chess-player-from-data-gathering-to-data-augmentation-d51f471a61b8