Developing The Unbeatable Chess AI

In 1997, IBM's supercomputer 'Deep Blue' was the first Chess AI which made use of Deep neural network learning techniques to beat a world champion, Garry Kasparov. What fascinated me most was one of the two things: one being what it takes to create a successful Chess AI and why was it as difficult as it was to beat a human player, even if they were a world champion. This essay explores the process of creating the 'unbeatable' chess AI. At the end of the essay I have also attached a video showing my own chess model I have developed using the techniques discussed in this essay.

A neural network is a type of computer program that tries to solve problems by accepting inputs, processing data in a way that's similar to how our brains work and producing an output. There are several types of neural networks but for the purpose of training a chess engine, a convolutional neural network (CNN) is going to be used. CNNs are a type of neural network (see Figure 1 for a sample CNN architecture) which are specialised to handle image processing and recognition such as processing an image of a given chessboard state. They go about this process by scanning the image and then analysing the image for its patterns and features.

Like other neural networks, a CNN is composed of interconnected "neurons" that work together to learn from data and make decisions. Think of it like a complex mathematical function that can be trained to find patterns in data and make classifications. Each neuron is assigned a weight, which is a parameter that can be used to describe the strength and position of each neuron. Along with weights, neurones also have assigned biases which are offset values which control when the state of the neuron changes to activate it. To start with, the weights and biases are set to random values which as you might guess results in a poor performance. But the general concept is that we train our CNN through large chess board image data sets and our engine will improve and learn how to play chess according to the 'response' we give it in relation to the move it has played and the corresponding best move it should have played. As a result, the CNN will tune its weights and biases based on our feedback to improve its performance for the next time it encounters the same game state by playing a better or even the best move.

The first stage is to create a working GUI chess two player game which successfully complies all the rules of chess. Following that is the collection of large image data sets of chess game states which also have attached the best move made by a super grandmaster for that particular game state so we can teach the CNN to play that best move.

To be able to recognise a particular chess game state, the CNN will need to process the image and analyse the game state so that it can recognise that game state in the future and pick the best move it has learned.

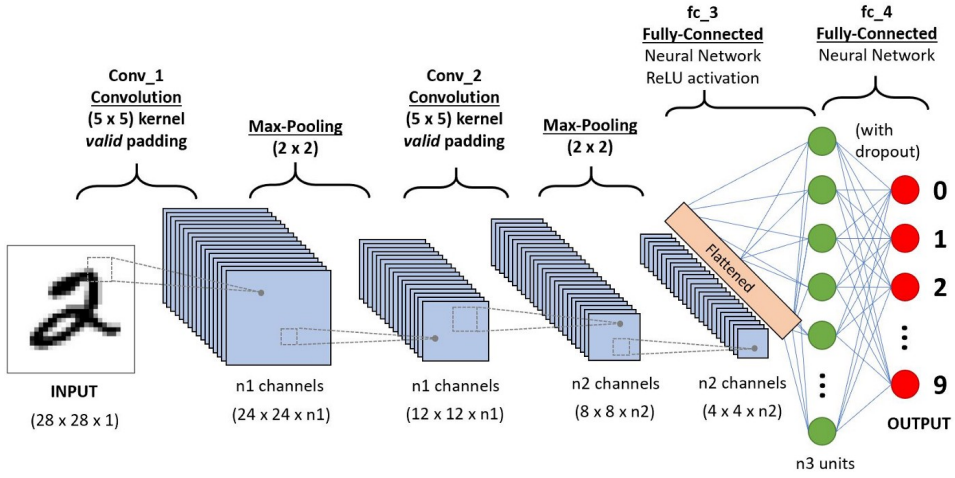

Figure 1: A typical CNN architecture [1]

First the image passes through a convolutional layer which applies a matrix filter to the image to sample a section of the input image. These filters are 'learned' meaning they're designed to extract features such as its dimensions, position of bishops, pawn count etc.

Following that is the pooling layers which are used to summarize information about the board state after applying a convolutional filter. For example, the output of a convolutional layer might represent the presence of certain types of pieces in certain positions on the board. The pooling layer could then summarize this information by selecting the maximum or average value in each sub-region of the output. In the context of our CNN engine, the information is a series of vector values which represent mainly the playable moves as well as other features such as the position of pieces.

The final output layer converts these vectors of real numbers into vectors of probabilities summing to 1 using something known as the SoftMax function, where each probability corresponds to the probability of playing a particular move. The formula for this probability distribution is defined as:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Equation [1]

This formula takes the natural exponent of the *i* th element (move) and divides it by the sum of the natural exponents of all the elements, and then returns the result.

In Eq. (1), $\vec{z}$ represents the input vector which represents a playable move.

The numerator e^{z_i} applies an exponential function to each element of the vector. This value can range from being extremely close to 0 for negative exponents to extremely large for large

positive exponents, so would lead to an invalid probability distribution as the probabilities wouldn't sum to 1. The solution to this is the denominator which acts to normalise the term by scaling the probability to a range of 0 to 1 thus allowing a valid probability distribution. To see why this formula produces a valid probability distribution, consider what happens when you apply it to a vector of arbitrary real numbers. Since the exponents are always positive, the numerator will always be positive, and the denominator will always be positive as well. Therefore, the output of the SoftMax function will also be positive.

Furthermore, since the denominator $\sum_{j=1}^{K} e^{z_j}$ is always greater than or equal to the numerator, the output of the SoftMax function will always be less than or equal to 1. Finally, since the denominator is the sum of all the exponents from j which is the index of the first move and K which is the last move , the output of the SoftMax function will always sum to 1. Therefore, the SoftMax function produces a valid probability distribution over the elements of the input vector.

Key reason we choose to use a probability distribution rather than a scoring system for each move as it is considerably more effective when choosing the move to play in cases where there are multiple good moves to be played or the best move is not easily apparent. A scoring system is much more likely to assign the same value to two or more moves in comparison to assigning a probability to each move thus hindering the engines decision-making.

So currently, we can process a game state and convert a particular move into a vector real number that the engine can understand with the vector value mapping to a probability of playing that move. To improve a neural network's performance, we need to be able to measure its performance. This is done using a value known as the 'loss' which represents the inaccuracy between an expected result and the actual outcome – the greater the error, the greater the loss. In the context of our chess engine, the loss function calculates the error between the expected best move to be played for a given board state and the actual move the AI played. There are many different types of loss functions that can be used in a chess CNN, but one common choice is Mean squared error.

$$MSE = (1 - Ypred)^2 + 2(Yactual)^2$$

Equation [2]

For example, if the engine plays a move with a probability of 0.1 and the probability of the best move to be played is 0.2. This error can be calculated as the sum of the probability of the actual move squared multiplied by 2 and the square of the complement of the probability of the best move. For this example:

$$loss = (1 - 0.2)^2 + 2(0.1)^2 = 0.66$$

A loss of 0.66 indicates that our network is performing poorly for the current assigned weights therefore we need to continue to train the model. So, let's say we train the CNN with 1 set of

games, we can then store each loss value for every move played and at the end of the set, we can calculate the mean loss to give an overview on the current performance of the engine. The greater the average loss, the worse the engine. To improve our engine further, we need to minimise the loss function.

The way the CNN evaluates moves and calculates their vector probabilities are all dependent on the assigned weights. In order to minimise the loss, we need to find the weight which minimises the loss and assign that particular weight.

What we do is that we define a loss function in terms of weight and for each game state, we vary the weight parameter and plot a graph of loss against weight. We don't just calculate the loss values for a large range of weights as this is highly inefficient so instead, we start at an arbitrary weight, and we move to its local minimum. This is done using a method known as gradient descent.

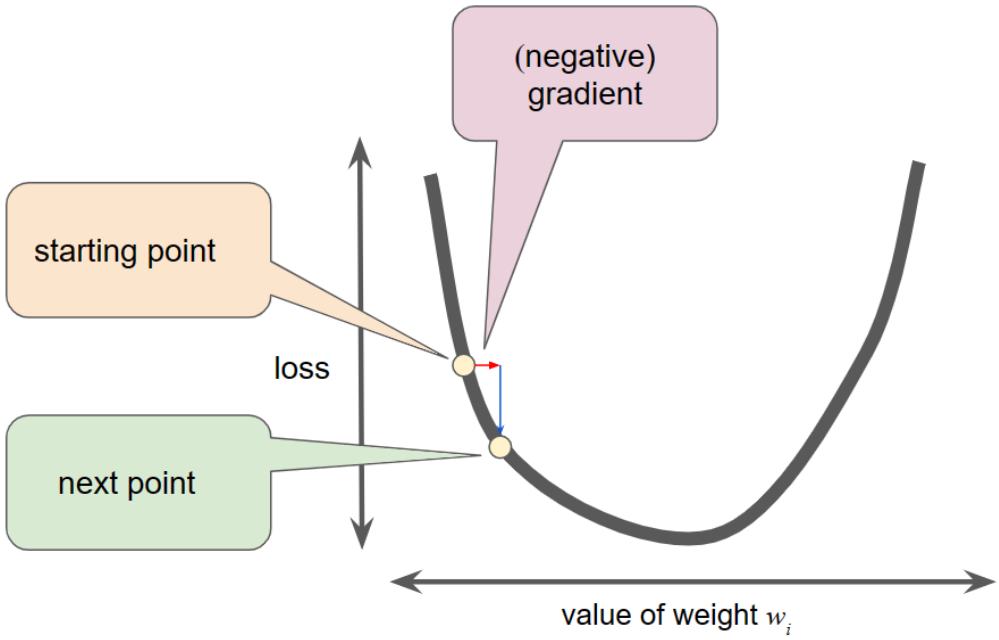

Figure 2: Example Loss-Weight Function [2]

The gradient of the loss curve is calculated at this point. The gradient is a vector so has a direction and magnitude. As the gradient always points in the direction of the steepest increase in loss, the gradient descent algorithm works by taking a step in the direction of the negative gradient to minimise loss at the quickest rate. To determine the next point on the curve the gradient descent algorithm moves to the side relative to the magnitude of the gradient i.e., if the point was already close to the minima, it would need to be moved a smaller horizontal distance identified by its shallower gradient than a point towards a maxima with a larger gradient and needs to take a larger step. So, if the gradient is negative, the point takes a step right and down to find the local minimum. If the gradient is positive, the point takes a step left and down to find the local minimum. The gradient descent iterates getting closer to the local minimum each time.

However, the issue with this method is that there are several million variables that affect the loss value. Using the above method would result in a graph with millions of dimensions due to the millions of parameters. To solve this, we need to slightly adjust the gradient descent algorithm.

To minimise the loss function, we can calculate the partial derivatives of the loss function. Partial derivatives are used to describe how much a function changes when one of its inputs changes, holding all other inputs constant. In the case of machine learning models, we want to know how much the loss function changes when we change each of the model parameters. This means calculating the gradient of one parameter e.g., weight while keeping every other parameter constant e.g., bias.

As our model shall be going through the gradient descent algorithm billions of times, it is unrealistic to compute the partial derivatives as a human would, so the machine uses a method known as reverse mode automatic differentiation.

The key idea behind reverse mode automatic differentiation is to use a computational graph to break down the function into a sequence of elementary operations. The general model of automatic differentiation is represented as a graph (see Figure 3) where the nodes represent arithmetic operations and functions such as ln (). Each edge represents the flow of data between operations. The inputs to the function are represented as the leaves of the graph, and the outputs are represented as the root.

The inputs are passed through in the forward pass and evaluated and then in the backward pass the gradients of the function with respect to the inputs are estimated by propagating the gradients backwards through the graph. We compute the gradients of the loss function with respect to the input network parameters, using the chain rule. The chain rule tells us that the derivative of a composite function can be computed by multiplying the derivatives of its individual components, and this is what allows us to compute the gradients of the loss function with respect to all the parameters of the network.

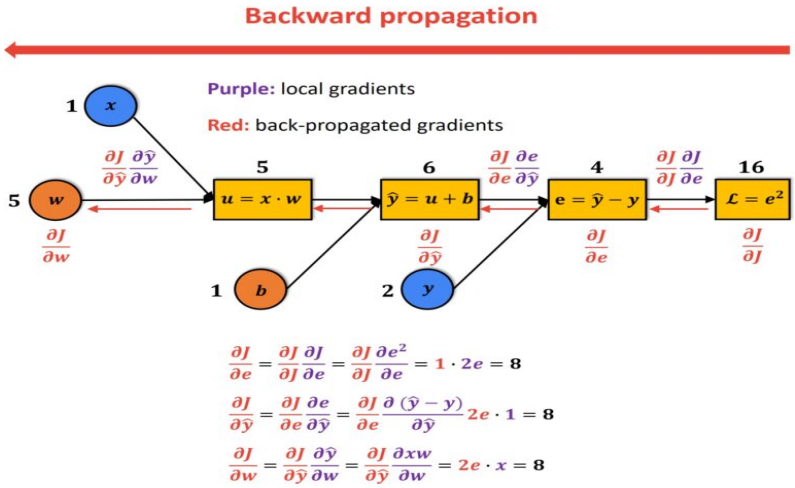

Figure 3: Example computational graph [3]

Using the gradients of the loss function with respect to all the parameters of the network, we can tune the weights and biases to achieve the minimum of the loss function and if all is correct, the network's performance shall improve and the average loss shall fall as more training sets are taught overtime, similar to the graph shown in Figure 4.

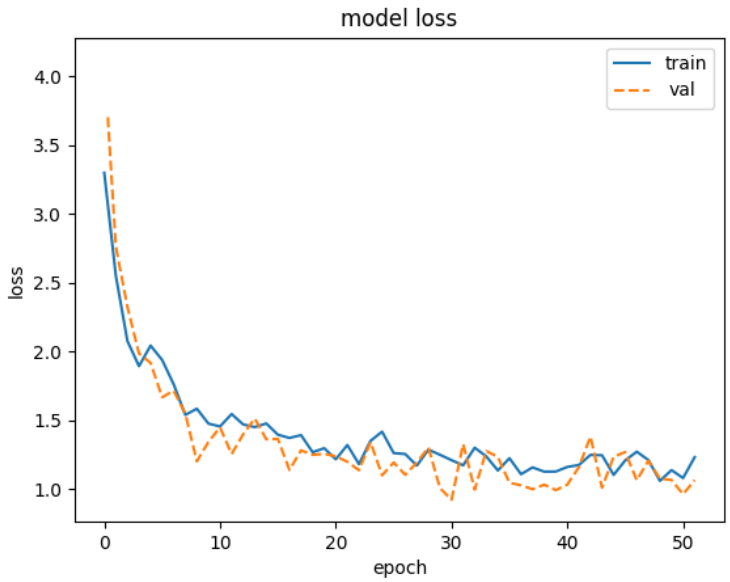

Figure 4: Typical graph of average loss vs epoch [4]

Overall, the development of a smart chess AI requires a combination of machine learning techniques and programming skills to design an engine that can compete with human players. The processing power required is large but with the growth of inducted processing power over the cloud and the vast range of millions of chess games, the potential for training a CNN model is endless with the model becoming better at identifying the 'perfect' move with greater accuracy significantly faster. It is a challenging task, but the results can be truly impressive, as demonstrated by IBM's Deep Blue, which beat a world champion, Garry Kasparov, in 1997.

Below is a video link of me playing a simplified neural network chess engine I have developed using techniques such as automatic differentiation, SoftMax Function and gradient descent algorithm. It is composed of 5 layers and is developed using modified data sets from lichess.com. The approximate distribution of the data is 30% Training, 50% Prediction and 20% Testing. Although it isn't probably as good as DeepBlue, it is still pretty fun to play with :)

Copy and paste link in a browser:
https://drive.google.com/file/d/14ScVEEgVs8-hXfcG-YxY4LcP913lVq4G/view?usp=share_link

References:

[1] An Introduction to Gradient Descent and Backpropagation | by Abhijit Roy | Towards Data Science

[2] Reducing Loss: Gradient Descent | Machine Learning | Google Developers

[3] Reverse-mode automatic differentiation: a tutorial - Rufflewind's Scratchpad

[4] https://towardsdatascience.com/automatic-differentiation-explained-b4ba8e60c2ad

[5] Convolutional Neural Networks, Explained | by Mayank Mishra | Towards Data Science

[6]What is a neural network? | TechRadar

Figure[1] - Convolution Neural Network (CNN) - Fundamental of Deep Learning - Idiot Developer

Figure[2] - Reducing Loss: Gradient Descent | Machine Learning | Google Developers

Figure[3] - https://datahacker.rs/004-computational-graph-and-autograd-with-pytorch/

Figure[4]- https://www.researchgate.net/figure/The-parallel-plots-of-loss-over-the-epochs-for-the-training-and-validation-sets-for_fig4_332645766