Chessbot

The Mathematics Behind The Neural Network

Hegyi Erik

February 17, 2026

1 Notation

Let $\dim(\mathbf{T})$ be a function, which returns the dimensions of the $\mathbf{T}$ tensor, or a layer in a neural network.
Let $\text{len}(\mathbf{v})$ be a function, which returns the length of the $\mathbf{v}$ vector.
Let $\mathbb{B}$ be the binary set, where $\mathbb{B} = \{0, 1\}$
Letting $\mathbf{T}$ be a general, N-th order tensor, we are going to define 2 functions:

1. **Summation**
   A summation symbol without start and end indices is going to mean summing each element of the tensor:

$$\Sigma(\mathbf{T})$$

2. **Product**
   A product symbol without start and end indices is going to mean multiplying each element of the tensor:

$$\Pi(\mathbf{T})$$

Letting $\mathbf{F}$ be an array of functions, where:

$$\mathbf{F} = \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{pmatrix}$$

We are going to define a shorthand notation for creating composite functions from this array, where:

$${}_i\mathbf{F}(x) = f_i(f_{i-1}(\ldots(f_1(x))))$$

And, if we want to create a composite function using all the functions:

$$\mathbf{F}(x) = {}_n\mathbf{F}(x) = f_n(f_{n-1}(\ldots(f_1(x))))$$

2 Representation of the Chessboard

2.1 Notation

Let P be the number of different pieces on a chessboard. On a normal chessboard, $P = 6$, since there are 6 different pieces:

- King
- Queen
- Rook
- Knight
- Bishop
- Pawn

Let Z be the number of players on the chessboard. In a normal game of chess, there are 2 players, so $Z = 2$, but there are variants with more players.

Since we are trying to generalize the notation, let D be the number of dimensions that the game of chess is played in. On a normal chessboard, $D = 2$, but there also exists a version of chess, which is 3-dimensional. It is impossible for humans to visualize higher dimensions, but theoretically, computers could learn to play chess in more dimensions. But for now, let's restrict ourselves to 2 and 3 dimensions.

Let $\mathbf{N}$ be the vector containing the dimensions of the chessboard, where:

$$\mathbf{N} \in \mathbb{N}^D$$

On a normal chessboard, where $D = 2$, $\mathbf{N}$ is going to be:

$$\mathbf{N} = \begin{pmatrix} 8 \\ 8 \end{pmatrix}$$

In other words, a normal game of chess is played in 2 dimensions, on a board which has dimensions of 8×8.

Now, let $\mathbf{T}$ be the tensor representing the state of the game, or in short, the **state tensor**. This tensor has to include every information about the board:

- The dimensions the game is being played in
- The dimensions of the board
- The number of players
- The number of pieces
- The positions of the pieces

This is how we are going to be keeping track of the game.

2.2 State Tensor

Let's start by representing the players on the chessboard. Since we have Z players, we can represent the state tensor as a vector of length Z, where each each element contains all the necessary information about one player:

$$\mathbf{T} = \begin{pmatrix} \text{player}_1 \\ \text{player}_2 \\ \text{player}_3 \\ \vdots \\ \text{player}_Z \end{pmatrix}$$

From this, the state tensor's dimensions would be:

$$\mathbf{T} \in \mathbb{R}^{Z \times \dim(\text{player})}$$

In other words, the state tensor is actually just a vector containing the **player tensors**, where the i-th player tensor contains all the information about the i-th player's position on the chessboard. The player tensors have to represent all the pieces of the player. Thus, we can write them as vectors of the **piece tensors**, where a piece tensor contains all the information about where a certain **type** of piece is. For example, the pawn's piece tensor would contain all the information about where **each** pawn is located.

$$\text{player} = \begin{pmatrix} \text{piece}_1 \\ \text{piece}_2 \\ \text{piece}_3 \\ \vdots \\ \text{piece}_P \end{pmatrix}$$

Thus, the dimensions of the player tensor are:

$$\text{player} \in \mathbb{R}^{P \times \dim(\text{piece}_i)}$$

To represent where a piece is located, we could store its coordinates, but that would mean that the model would have to learn what these coordinates mean, and what adjacency means, which would be difficult and costly. Instead, we are going to create a **binary map** of dimensions $N_1 \times N_2 \times \ldots \times N_D$, which is going to represent whether the i-th piece exists at the given coordinate, or not:

$$\text{map}_i \in \mathbb{B}^{N_1 \times N_2 \times \ldots \times N_D}$$

Now, we have represented the amount of players we have, and where each piece is located. But we have one thing left to do: represent the chessboard itself. In a normal game of chess, the board is a perfect square, but we are trying to generalize the model, so it can be trained on variations later on. Thus, we need a **binary mask**, which is going to state whether a tile is legal or not:

$$\mathbf{M} \in \mathbb{B}^{N_1 \times N_2 \times \ldots \times N_D}$$

Adding this extra dimension to the tensor is a surprisingly difficult problem, since, depending on where we add it, the model could interpret it as an extra dimension, an extra player, or an extra piece. Because of this, we are going to flatten the first two dimensions (players and pieces) of our tensor into one dimension of shape players $\times$ pieces. Then, we are going to add an extra dimension for our mask here. No data is lost, just dimensions compressed. Once we start training the model, it is going to have no problem learning that the first 6 pieces (in a normal game of chess) can not attack eachother. Thus, the dimensions of our final state tensor look like this:

$$\mathbf{T} \in \mathbb{B}^{(Z \cdot P + 1) \times N_1 \times N_2 \times \cdots \times N_D}$$

As a sanity check, let's substitute a normal game of chess' properties:

- $Z = 2$
- $P = 6$
- $\mathbf{N} = \begin{pmatrix} 8 & 8 \end{pmatrix}$

Thus, we get that for a normal game of chess, the state tensor is going to be of dimensions:

$$\mathbf{T} \in \mathbb{B}^{13 \times 8 \times 8}$$

Or, if flattened into a vector, it would contain 832 pieces of data.

3 Model design

3.1 Output

We want the model to predict the best move. However, this would be insanely complicated to do, if we actually wanted it to output something like `BxF4`. Instead, we are going to generate an array of every legal move, theoretically perform them, and rate these theoretical positions with the model. After this, we are going to select the best performing one - in other words, the move which is going to bring us closest to victory. This means that we only need the model to rate the position on the game. We are going to do this by predicting the winner of the game. Thus, the position can have the following values:

- White is winning: value $= 1.0$
- White is leading: $0.0 <$ value < 1.0
- Draw: value $= 0.0$
- Black is leading: $-1.0 <$ value < 0.0
- Black is winning: value $= -1.0$

This means that whatever the output is, we must normalize it, so the end result is between -1 and $+1$. The clear winner for this task is one of the most popular activation functions: $\tanh(x)$. The hyperbolic tangent function is perfect for this task, since it can take in any input, and it will output a number between -1 and $+1$, making it perfect for our task.

This value is not going to tell us the exact probabilities of one side winning, the hyperbolic tangent function is not capable of that, but intuitively, we can look at it that way.

3.2 Loss

For the loss, we are going to use mean squared error (MSE). For this, we need the predicted and the actual outcome. This means that we can only backpropagate and optimize **after** the game has ended, which fits pretty well with how chess masters train, analyzing each game after the game has ended. Let $\hat{y}$ be the actual outcome of the game, where

$$\hat{y} \in \{-1,\, 0,\, +1\}$$

Let y be the predicted outcome of the game, where

$$-1 \leq y \leq +1$$

Using these two variables, we can calculate the loss (Λ):

$$\Lambda(y, \hat{y}) = (y - \hat{y})^2$$

But it is easy to see two major flaws with this structure. The first one is that mistakes can occur in the game. One side may be winning, and the model would correctly evaluate the position as winning, but then a blunder could occur. If we only looked at the end result, then the model would be punished for not predicting an unpredictable bliunder, making training noisy and inefficient. We need the model to assume that that the best moves are played, and it should not think about individual playing styles or blunders.

The second problem is that the position is equal at the start of a gam. If we only looked at the end result, then the model would try to predict which side is going to win at the first move, even though it should say that the positions are balanced. It will have no way of predicting this correctly, and it is going to get punished, once again making training noisy and inefficient.

But we have a modification, which could solve this problem, called **bootstrapping**. We are going to predict who is most likely to win at the **next** step, and work our way backwards. This works on the assumption that the current position is going to be similar to the next one, which, once again, assumes no blunders. This is why it is going to be extremely important to train our model only against itself, so it plays "perfectly", and not against humans, who can make mistakes. Of course, in the beginning our model is also going to make mistakes, but over time the number of these mistakes is going to be reduced.

Letting $\mathbf{S}$ be the array containing the steps, and s be the number of steps (so $s = \text{len}(\mathbf{S})$), we are going to work our way backwards in the following way:

1. Step s: This is the last step - either the checkmate, or the step triggering a draw. Thus, we know exactly which side won (if any). This is going to be the end value: $\hat{y} \in \{-1, 0, +1\}$

2. Step $s-1$: We are going to predict the state of the game at step $s-1$. Let's call the prediction y_{s-1}, and the target value $\hat{y}$. From this, we can compute the loss: $\Lambda = (y_{s-1} - y_s)^2$

3. Step $s-2$: We are going to predict the state of the game at step $s-2$, which is going to be our prediction y_{s-2}. Then, our target value is going to be the predicted position at step $s-1$, meaning that our loss is going to be: $\Lambda = (y_{s-2} - y_{s-1})^2$

Generally, our loss function at step i is going to be:

$$\Lambda_i^{BS}(y_i, y_{i+1}, \hat{y}) = \begin{cases} (y_i - y_{i+1})^2;\ i+1 < s \\ (y_i - \hat{y})^2;\ i+1 = s \end{cases}$$

There is another, simpler way to define a loss is the **Monte Carlo** method. This works by looking **only** at the end result of the game ($\hat{y}_s$). This is the sort of loss function which we originally described, and then listed its shortcomings. But, combined with the bootstrapping method, it could be useful, especially since the bootstrapping method has shortcomings of its own (for example, it can propagate errors). The Monte Carlo method compares the current position with the end position, making the loss function easy to write mathematically:

$$\Lambda_i^{MC}(y_i, \hat{y}) = (y_i - \hat{y})^2$$

We are going to define a hyperparameter, α, which is going to describe which method is preferred by the model. Thus, our final loss is going to look like:

$$\Lambda_i(y_i, y_{i+1}, \hat{y}) = \alpha\Lambda_i^{BS}(y_i, y_{i+1}, \hat{y}) + (1-\alpha)\Lambda_i^{MC}(y_i, \hat{y})$$

3.3 Layers

Our neural network is going to have many layers. For full generalization, this needs to be adjustable, so the exact number is not important in this generalized mathematical derivation. What is important, however, is what are the different **layer types** that the network is going to contain.

First of all, we need to define our notation:

- Let $\mathbf{L}$ be an array containing the layers.
- Let $l = \text{len}(\mathbf{L})$ be the number of layers.
- Let $\mathbf{X}_i$ be the input of the i-th layer, where $\mathbf{X}_1$ is the input of the model.
- Let $\mathbf{Y}_i$ be the output of the i-th layer, where $\mathbf{Y}_l$ is the output of the model. Usually, $\mathbf{X}_{i+1} = \mathbf{Y}_i$

Since a layer in a neural network is essentially a mathematical function, our array of layers is going to be an array of mathematical functions, each influencing the end result. The entire neural network is going to be one large composite function, meaning that the end result can be denoted as:

$$y = \mathbf{Y}_l = \mathbf{L}(\mathbf{X}_1)$$

Now, we can turn our attention towards the different types of layers we are going to use. First of all, we are going to define two layer types:

1. **Learning Layers**
   These layers are going to be the main source of learning for the neural network. They are not going to change the board's dimensions, since that would make the model lose very important knowledge. Instead, they are going to make the model learn by either not changing the dimensions at all, or changing only the channel dimensions, and not the spatial dimensions.

2. **Dimension Reductional Layers**
   These layers are going to be placed at the end of the network. Their purpose is to convert the many-dimensional tensor input into a single scalar output.

For the sake of convenience in notation, we can treat these two different layer types as different models. The first model, whose layers are going to be denoted as $\mathbf{L}^1$, and whose length is going to be $l_1 = \text{len}(\mathbf{L}^1)$ is going to responsible for learning the relationships between the pieces, and evaluating the position, while the second model, whose layers are going to be denoted as $\mathbf{L}^2$, and whose length is going to be $l_2 = \text{len}(\mathbf{L}^2)$ is going to be responsible for collapsing the dimensions, and outputting a single scalar number. Of course, when programming the model, we are going to put everything into one model, but for now, we are going to simplify the notation by splitting the model in two.

3.3.1 First Model

In our first model, each layer is going to use the ReLU activation function.

For our chess engine to work properly, the dimensions of the board must not change, otherwise important information would be lost. Let's remind ourselves what the input dimensions were:

$$\mathbf{T} = \mathbf{X}_1^1 \in \mathbb{B}^{(Z \cdot P+1) \times N_1 \times N_2 \times \ldots \times N_D}$$

Let us represent the channels as $C_1 = Z \cdot P + 1$, and the dimensions as $H_1 = N_1 \times N_2 \times \ldots \times N_D$ It is important to note that this does not mean multiplying the dimensions, and flattening into a vector. This is just a shorthand notation. Thus, we can write our input's dimensions as:

$$\mathbf{X}_1^1 \in \mathbb{B}^{C_1 \times H_1}$$

Or generally:

$$\mathbf{X}_i^1 \in \mathbb{B}^{C_i \times H_i}$$

With this notation, we can easily mathematically describe the rule that the dimensions of the board can not change:

$$H_1 = H_2 = \ldots = H$$

This means that we are still going to have a $D+1$-dimensional tensor, while we need a 0-dimensional scalar. This is going to be the job of our last two layers: converting the tensor into a scalar.

Now, since we are using a convolutional neural network, we need a **kernel** for each layer. This is going to be the **weight tensor** of the particular layer. The size of the kernel in layer i is going to be:

$$\mathbf{W}_i^1 \in \mathbb{R}^{C_i \times C_{i+1} \times k_i^1 \times k_i^2 \times \ldots \times k_i^D}$$

And the bias of the layer is going to be b_i.

Now, we are going to map the input of the layer to the output of the layer, using the kernel. Let

$$\mathbf{a} \in \mathbb{R}^D$$

be a vector representing the point we are working with, where

$$\forall j \in \mathbb{N}; 0 < j \leq D : 0 < \mathbf{a}_j \leq \mathbf{N}_j$$

This way, for each point $\mathbf{a}$, we can calculate the output before applying activation function. Letting $\mathbf{X}_i$ be the incoming input, we can write the output as:

$$\mathbf{Z}_i^1(\mathbf{a}) = b_i^1 + \sum_{j_1=1}^{k_i^1} \dots \sum_{j_D=1}^{k_i^D} \sum_{c=1}^{C_i} \mathbf{W}^1(j_1, j_2, \dots, j_D, c) \cdot \mathbf{X}_i^1(\mathbf{a}_1 + j_1, \dots, \mathbf{a}_D + j_D, c)$$

Or, using vector notation:

$$\mathbf{Z}_i^1(\mathbf{a}) = (\mathbf{W}_i^1 \cdot \mathbf{X}_i^1)_{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_D} + b_i^1$$

Then, we just need to apply the activation function to get the final output:

$$\mathbf{Y}_i = \mathrm{ReLU}(\mathbf{Z}_i^1)$$

Now that we have defined our base layers, we can move on, and define the other type of layer, namely the **residual block**. Residual blocks are really powerful in chess engines, because they allow the model to learn how to modify the values, without needing to remember them. It works the same way as a base layer, except, at the end, we add back the input:

$$\mathbf{Y}_i^1 = \mathrm{ReLU}(\mathbf{Z}_i^1) + \mathbf{X}_i^1 \iff \dim(\mathbf{L}_i^1) = \dim(\mathbf{L}_{i-1}^1)$$

This is so similar to the base layer that we can combine the two expressions into one:

$$\mathbf{Y}_i^1 = \mathrm{ReLU}(\mathbf{Z}_i^1) + \eta_i \mathbf{X}_i^1$$

Where:

$$\eta_i = \begin{cases} 1 & \text{if residual block} \\ 0 & \dim(\mathbf{L}_i^1) \neq \dim(\mathbf{L}_{i+1}^1) \\ 0 & \text{otherwise} \end{cases}$$

3.4 Second Model

We are going to use **Global Average Pooling** (GAP) to turn our $C_{l_1} \times H$-dimensional tensor into a C_{l_1}-dimensional vector. GAP works by looking at each channel, and averaging all the values inside that channel, effectively reducing the board dimensions in each channel to one scalar number:

$$\mathbf{Y}_1^2 = \mathrm{GAP}(\mathbf{X}_1^2) = \frac{1}{\Pi(\mathbf{N})} \begin{pmatrix} \Sigma(\mathbf{X}_{11}^2) \\ \Sigma(\mathbf{X}_{12}^2) \\ \vdots \\ \Sigma(\mathbf{X}_{1C_{l_1}}^2) \end{pmatrix}$$

Where:

$$\mathbf{X}_1^2 = \mathbf{Y}_{l_1}^1$$

This collapses the tensor into a vector, allowing us to make a prediction on the next layer:

$$y = \mathbf{Y}_2^2 = \tanh(\mathbf{W}_1^2 \cdot \mathbf{X}_2^2 + \mathbf{B}_2^2)$$

3.5 Current player

The model evalutes white's position, but we need it to be able to evaluate any player's position. Thus, when evaluating a player's position, who is not `Player 1`, we need to "flip" the board. This is not a mathematical structure, thus it is not going to be defined here, since this document only deals with the mathematical structure of the neural network, but it is important to realize that the state vector is going to undergo some transformations depending on whose turn it is. Further explanation will be written when the programming design of the engine is discussed.

4 Backpropagation

This document only deals with the pure mathematical representation of the model, and, to efficiently execute backpropagation, we need things like memory usage, and other things only available in programming. Thus, a new document is going to be opened purely for backpropagation.