The Math Behind Neural Networks

Josh Hess

December 10, 2024

Introduction

This document is a transcription from the YouTube video Building a neural network FROM SCRATCH (no Tensorflow/Pytorch, just numpy & math) by Samson Zhang. I wanted to create my own document going over the mathematics behind how a multilayer perceptron works to go along with my notebook containing the code. The math I cover here is owed to Samson Zhang, I linked to a video that was a massive help during this project.

The dataset we will be working with is the MNIST dataset. It contains 70000 images of handwritten digits, with 60000 in the training set and 10000 in the testing set. For our *input* data, we will have images that are 28×28 pixels, 784 pixels total.

We will be using a *multilayer perceptron* model, which is a basic neural network. It is composed of *nodes* that typically contain some value between 0 and 1.

For our network, we will have the same layout as Figure 1. The input layer (the $0th$ layer) will contain 784 nodes, one for each of our input pixels. Each

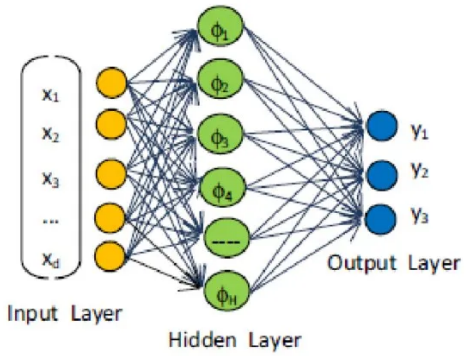

Figure 1: Multilayer Perceptron

pixel will have some value x_i between 0 and 255 representing the "activation" of that pixel, which can be interpreted as how bright the pixel is in the image. The hidden layer (the $1st$ layer) will have 10 nodes, as will the output layer (the $3rd$ layer). The hidden layer will help with training, but we'll get into the specific mathematics later. The output layer has nodes representing each of the 10 possible digits $1 - 9$.

Now that we have a foundation of what we're doing, let's get into the mathematics.

Basic Definitions

For our inputs, we'll load all of our example images into a matrix X containing one row for each image. Each column represents the value of one of the pixels. We'll begin by taking the transpose of this matrix, so that each column is one image, and each row represents one pixel:

$$X = \begin{bmatrix} — & x^{(1)} & — \\ — & x^{(2)} & — \\ & \vdots & \\ — & x^{(m)} & — \end{bmatrix}^T = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

where m is the number of images used for our neural network. Now, we'll go into each of the main steps of our neural network.

Forward Propagation

Forward propagation is the algorithm that populates our neural network with new values for a new prediction. Whenever we run a new training cycle on our network, we will run forward propagation again with new values. First, we have our input layer:

$$A^{[0]} = X \ (784 \times m)$$

This gives us the values for our first 784 nodes in our input layer. Now, we have to populate the *unactivated* values for the hidden layer, called $Z^{[1]}$. This is calculated as a dot product between our input layer nodes and all of the possible connections (denoted as *weights*) between the input layer nodes and hidden layer nodes. We also add another element to this product called a *bias*, which can help us further tweak the network. This is represented with the following equation:

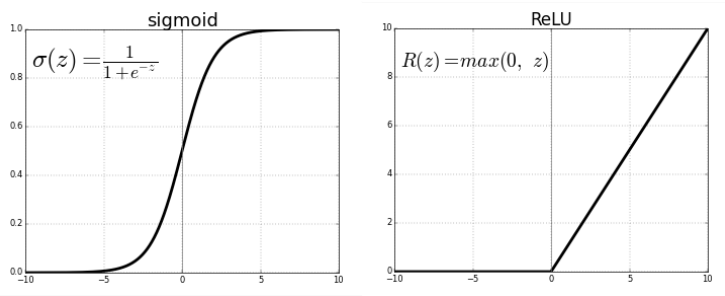

Figure 2: Sigmoid and ReLu Functions

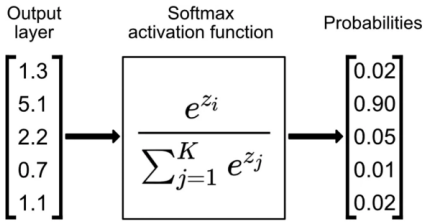

Figure 3: Softmax Function

$$Z^{[1]} = w^{[1]} A^{[0]} + b^{[1]}$$

where $w^{[1]}$ is the weight vector and $b^{[1]}$ is the bias vector. Now, these nodes are not very useful to us unactivated. We are essentially calculating a linear regression with some fixed error (the bias). To add a layer of complexity to our model, we can use an *activation function*. Two of the most common types are the *sigmoid* and *ReLu* functions. (Figure 2)

Both functions serve to essentially "squish" the values onto another scale. Both functions work, but we will be using the ReLu function for our activation function. This is because the ReLu function is very computationally efficient compared to other activation functions, making it easy to work with. To calculate our hidden layer $A^{[1]}$, we can use the following equation:

$$A^{[1]} = \text{ReLu}(Z^{[1]})$$

Now we're ready to move onto the output layer $A^{[2]}$. We will do a similar operation at first to calculate our unactivated output node values:

$$Z^{[2]} = w^{[2]} A^{[1]} + b^{[2]}$$

where $w^{[2]}$ represents the weights between the hidden and output layers, and $b^{[2]}$ represents the bias for the output layer. These weights and biases

are different from the ones used to calculate the unactivated hidden layer $Z^{[1]}$. Then, we need to apply a different type of function.

We essentially want each node in our output layer to represent a *probability*. This is the probability that the digit in the image is that particular number, being a digit between 0 and 9. To do this, we can use the Softmax function as defined in Figure 3. This function calculates the ratio of some e^{z_i} to the sum of all e^{z_j}. In other words, It creates a probability distribution for our output layer.

Using this Softmax function, we can calculate our output layer:

$$A^{[2]} = \text{Softmax}(Z^{[2]})$$

Finally, the goal of our neural network is to predict which of the 10 digits is the correct one. So we will calculate our prediction $\hat{y}$ as $\max(A^{[2]})$. So in summary, our full forward propagation algorithm is as follows:

Input Layer

$$A^{[0]} = X$$

Hidden Layer

$$Z^{[1]} = w^{[1]}A^{[0]} + b^{[1]}$$
$$A^{[1]} = \text{ReLu}(Z^{[1]})$$

Output Layer

$$Z^{[2]} = w^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = \text{Softmax}(Z^{[2]})$$

Prediction

$$\hat{y} = \max(A^{[2]})$$

Backpropagation

Backpropagation is the main algorithm that drives the "learning" of our neural network. It allows us to calculate how far off each node was from where it should have been, and adjust all the weights as a result. It allows us to slowly improve the performance of our network over time. We'll start by defining $dZ^{[2]}$, our errors for the output layer:

$$dZ^{[2]} = A^{[2]} - Y$$

Where $A^{[2]}$ is the output layer from our forward propagation algorithm and Y is the correct output layer. We use *one-hot encoding* for the correct layer.

For example, if 4 was the correct digit, the node representing 4 would have a value of 1, while every other node would have a value of 0. (This is because if the digit is 4, in an ideal network that would be the only node that would activate)

Now we need to get into some fairly dense multivariate calculus. We will simplify notation where possible to make it more digestable. We need to quantify how much our weights and biases contributed to the loss function $dZ^{[2]}$. We can do this by calculating the derivative of the loss function with respect to the weights and biases respectively. The formulas for both are below, but understanding them is not necessary for the topic:

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$
$$db^{[2]} = \frac{1}{m} \sum dZ^{[2]}$$

The formula for $db^{[2]}$ is fairly simple. It's simply an average of the loss function $dZ^{[2]}$. Now, we need to do the same thing for the weights and biases for our hidden layer. We will use slightly different approach:

$$dZ^{[1]} = w^{[2]T} dZ^{[2]} \cdot \text{ReLu}'(Z^{[1]})$$
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$
$$db^{[1]} = \frac{1}{m} \sum dZ^{[1]}$$

The most important step above to highlight is the first. We are essentially doing forward propagation backwards (hence, 'backpropagation') by using the transpose of the weights and using the derivative of our activation function (in this case, ReLu[1])

Updating Parameters

Now that we've calculated our errors, we want to update our parameters before running forward propagation again. We will do this by subtracting our errors from our node values to calculate the new updated node values. We will multiply each error by a *learning rate* α. This is what's referred to as a *hyperparameter*, which is a parameter set by the researcher and not the algorithm itself. This doesn't have a "default" value, it's something that can be tweaked by the researcher to try and optimize the model's performance. These equations represent our parameter updates:

[1]ReLu is actually non-differentiable at the origin. Most computer algorithms that implement ReLu simply set the derivative equal to zero when it is undefined to avoid this issue, but it was worth pointing out.

$$
\begin{aligned}
w^{[1]} &:= w^{[1]} - \alpha \cdot dw^{[1]} \\
b^{[1]} &:= b^{[1]} - \alpha \cdot db^{[1]} \\
w^{[2]} &:= w^{[2]} - \alpha \cdot dw^{[2]} \\
b^{[2]} &:= b^{[2]} - \alpha \cdot db^{[2]}
\end{aligned}
$$

This completes our multilayer perceptron. Our training consists of a number of training *cycles*. Each cycle consists of

1. Forward Propagation (Calculate output layer $A^{[2]}$ and prediction $\hat{y}$

2. Backpropagation (Calculate errors from correct value)

3. Update weights and biases according to learning rate α

We repeat this for as many training cycles as we deem necessary until our model performs as we need it to. That concludes the mathematics behind our neural network. To see the code implementation, see the iPython notebook file MultilayerPerceptron.ipynb in the repository.