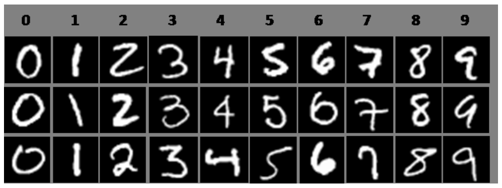
Exploring Neural Networks through a Mathematical Approach

Semih Gultekin

Introduction

With the advent of various up-and-coming artificial intelligence models, it is crucial to understand the foundation upon which they are built. Whether in healthcare or finance, most AIs are trained using deep learning or more specifically, neural networks. However, due to the complex nature of neural networks, creating one from scratch requires significant mathematical theories and techniques, especially linear algebra. Ranging from processing hundreds of data points to engaging in many matrix-based computations, the role linear algebra plays in neural networks is significant. In this paper, I wish to expose the importance and role of linear algebra within neural networks. Additionally, I aim to introduce the various pieces and methods of a neural network by walking through the process of creating a basic neural network from scratch.

The data that the model will train upon and create predictions is the MNIST data set. The MNIST data set is a collection of nearly 42,000 28 by 28-pixel images representing hand-drawn numbers from 0 through 9 (inclusive). The goal of the model is to recognize any number between 0 and 9 drawn on a 28 by 28-pixel image.



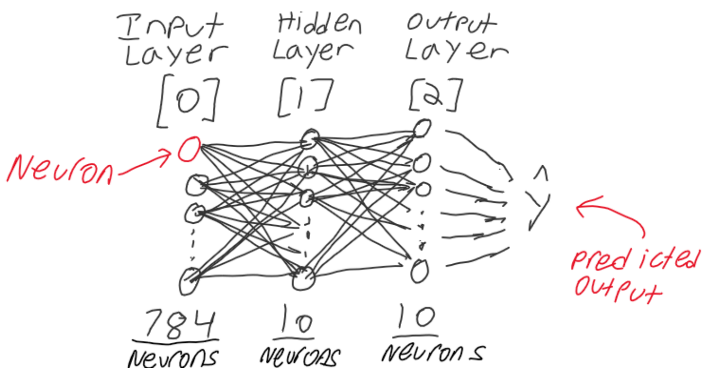
Example Image from MNIST Data Set

Anatomy of Neural Network

Neural networks are complex models with intricate layers and pieces. However, to create a network from scratch, it is necessary to understand everything from the ground up. Beginning with the most basic piece, a **neuron** (sometimes called a node). A neuron is a computation function that takes an input and processes it using **Weights** and **Biases** to produce a result. Weights and Biases help the neural network differentiate the importance of each neuron's result. The bulk of a neural network is neurons and they are organized in **Layers**. Layers are a collection of neurons that work together to serve a role within a neural network (IBM). The main layers are:

- Input layer: Receives data for training
- Hidden layer(s): Primary portion for processing and predicting data
- Output layer: Produces the final prediction(s) for the neural network

To help visualize the neural network that will be created, a drawing below is provided:



To add additional context to the image above, the input layer is taking a matrix with 784 entries with each neuron corresponding to an entry. Each entry is simply a pixel from a 28 x 28 image, with each pixel representing a number from 0 - 255 (0 being completely black, and 255 being completely white).

Mathematics and Methods

Up until this point, everything regarding the neural network has been explored through a high-level overview. In this section, we will dive into all of the math and actual functions/methods that make the neural network work.

The first step in a training iteration is to move through the entire network from left to right using a method called **Forward propagation.** In essence, forward propagation is the process of feeding input into the network and moving from layer to layer until the final output is produced (IBM). Below is the mathematical representation of the entire forward propagation method (Zhang):

For starters here are some variable definitions:

A^0 = Input layer (zeroeth layer) 784 pixels/one image

Z^1 = Hidden layer (first layer)

Z^2 = Output layer (second layer)

The first step is to format the training data (transposing if necessary) into a 784 x m matrix, making each column one image. Then setting A^0 equal to this training data matrix:

A^0 = X (X = 784 x m training data matrix)

*NOTE: The reason why the matrix is not 784 x 1 is to feed the training data all at once for efficiency instead of creating a loop and training one by one.

We then feed the input data into the first layer and complete the following equation:

w^1 = Weights for the first layer (10 x 784 matrix)

b^1 = Biases for the first layer (10 x m matrix)

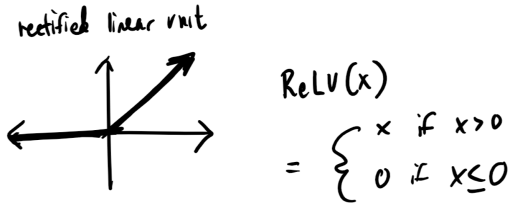
$Z^1 = w^1 \bullet A^0 + b^1$

In this equation we are doing a matrix multiplication between the weight matrix and the input layer and then adding biases to the resulting matrix.

*NOTE: For the first ever iteration, the weights and biases are initialized to random numbers between 0 - 1, but then in further iterations they are never re-initialized

---

So far we have only done a fancy linear combination through matrix multiplication and addition (Zhang). To combat this and add complexity we apply **Activation functions** to neurons. For this network, the rectified linear unit (ReLU) function will be used. Which states that for any input zero or less, it returns zero, otherwise it returns the original input.


rectified linear unit
ReLU(x)
$= \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$

Samson Zhang

We will apply this function to every neuron in the first layer:

$A^1 = \text{ReLU}(Z^1)$

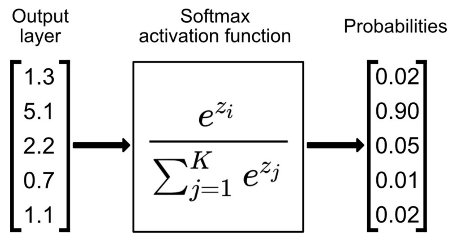
Now we will conduct the last few steps again for the second layer shown below:

w^2 = Weights for the second layer (10 x 10 matrix)

b^2 = Biases for the second layer (10 x m matrix)

$Z^2 = w^2 \bullet A^1 + b^2$

Finally, a new activation function called Softmax will be applied to every neuron in the second layer. Softmax will calculate the probabilities for a predication to be true and output them in a vector as shown below:


Output layer
Softmax activation function
Probabilities
$\begin{bmatrix}1.3\\5.1\\2.2\\0.7\\1.1\end{bmatrix} \rightarrow \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \rightarrow \begin{bmatrix}0.02\\0.90\\0.05\\0.01\\0.02\end{bmatrix}$

Samson Zhang

We apply this to our neural network through this equation and have hence, completed one full forward propagation through the network:

A^2 = SoftMax(Z^2)

Now that we have walked through a forward propagation, the network will provide some predictions. However, these predictions are near meaningless guesses without any real training or adjustments to weights and biases, which is where the **loss function** comes in. The loss function is used to determine the **loss value** which is the difference between the predicted output and the real answer. Based on the loss value, the weights and biases are tweaked in order to reduce the loss value through **Backward propagation.** Backward propagation is the method of moving backward from the output layer to the input layer tweaking weights and biases as we go. This entire process is often called **gradient descent** (IBM).

We will begin at the second layer and use a function called Onehot to encode the actual number the network is predicting into a 10 x 1 vector with all entries as zero except for the entry corresponding to the actual number which will be set as 1.

$$\text{Ex: } \hat{y} = 5 \,\Big|\, \text{onehot}(y) = Y = \begin{bmatrix}0\\0\\0\\0\\0\\1\\0\\0\\0\\0\end{bmatrix}\begin{matrix}0\\1\\2\\3\\4\\5\\6\\7\\8\\9\end{matrix}$$

This vector will then be used to subtract from A^2 (prediction probabilities) to get the layer two error value:

$dZ^2 = A^2 - Y$

We removed the prediction probability for the actual number in order to **not** change the weights and biases associated with it during tweaking as it was the correct prediction.

Using dZ^2, we can calculate the error for the layer two weights and biases through the following equations (Zhang):

$dw^2 = \frac{1}{m} dZ^2 * transpose(A^1)$

$db^2 = \frac{1}{m} \Sigma\, dZ^2$

We then move from the second layer to the first layer and begin calculating the error. However, this portion is a little complicated as we are conducting forward propagation in reverse. In essence, we apply the weights to the error from the second layer in order to get the error for the first layer.

Additionally, we multiply the **derivative of the activation function** to undo the original activation function so we can get proper error.

$$dZ^1 = transpose(W^2)\, dz^2 \ * \ g^{/}(z^1)$$

Finally, we do similar equations as before for the first layer weights and biases error:

$$dw^1 = \frac{1}{m} dZ^1 \ * \ transpose(X)$$
$$db^1 = \frac{1}{m} \Sigma\, dZ^1$$

Now the very final step to the entire process is to update the weights and biases:

α = learning rate given by the user (constant term), in this network 0.1 is used

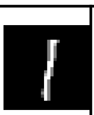
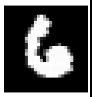
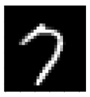
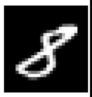
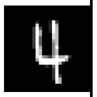
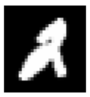
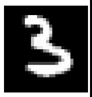
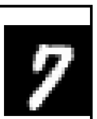
$$w^1 = w^1 - \alpha * dw^1$$
$$b^1 = b^1 - \alpha * db^1$$
$$w^2 = w^2 - \alpha * dw^2$$
$$b^2 = b^2 - \alpha * db^2$$

After updating the weights and biases, the entire process starting from forward propagation begins again. However, this time the weights and biases will yield *slightly* better results, and over time, the predictions will become more accurate and meaningful.

*NOTE: For the entire code implementation and to try the neural network yourself, it is on my GitHub repo: https://github.com/ppoptart12/NeuralNetworkFromScratch

Results

While not perfect, upon 500 iterations of gradient descent, the neural network reached approximately 85% (plus or minus 2%) accuracy. However, doubling the iterations to 1000 only brought the approximate accuracy to around 87% (plus or minus 1%). For visualization, there are a few of its predictions and attached images below.

											
Actual	1	6	5	2	7	8	4	2	3	5	7
Predict	1	6	5	2	7	8	9	8	6	7	9

Discussion

Although the approximate 85% accuracy is decent, it is *nowhere* close to perfect. In fact, there are many simple fixes that can be made in order to increase the accuracy of the network. For starters, drastically increasing the number of neurons per layer will add additional complexity to the network to aid its training process. On that note, adding more hidden layers will also increase complexity further and help the network's ability to recognize number patterns.

In addition to increasing the layers and neurons of a neural network, altering mathematical functions may result in better predictions. For our network, the functions used in the hidden layer such as the ReLU function, were used to keep the network simple. However, far more sophisticated and complex activation functions can be used to get better predictions. Similarly, the gradient descent can be tweaked and completely altered to make the model far more accurate and efficient. No matter what, there will always be a way to make the model better through higher complexity.

Limitations

While it is certainly possible to increase the accuracy of the network, physical limitations bar it from becoming perfect. Simply put, our computers (especially personal laptops) are not powerful enough to run massive and complex neural networks. Most large-scale AI research institutions and companies have huge facilities dedicated to training and running models and these still take *days* to train a large model. Our computing speed limits the mathematical complexity, layers, and neurons our neural networks can host. Hence, as our hardware and software optimizations increase, surely our ability to create and train sophisticated neural networks will follow.

Conclusion

Neural networks are an extremely powerful tool that can be utilized in almost any aspect of our lives. Such as computer vision models in autonomous vehicles or even facial recognition software used in healthcare. In the upcoming years, more and more use cases will become apparent as the capabilities of neural networks continue to increase and as they become more sophisticated. In essence, neural networks are *almost* limitless, and with enough complexity and training, there is a real possibility for them to make lasting impacts on our lives.

Bibliography

What Are Neural Networks? | IBM. https://www.ibm.com/topics/neural-networks. Accessed 10 Dec. 2023.

Building a Neural Network FROM SCRATCH (No Tensorflow/Pytorch, Just Numpy & Math). www.youtube.com, https://www.youtube.com/watch?v=w8yWXqWQYmU. Accessed 11 Dec. 2023.

Understanding the Math behind Neural Networks by Building One from Scratch (No TF/Keras, Just Numpy) | Samson Zhang. https://www.samsonzhang.com/2020/11/24/understanding-the-math-behind-neural-networks-by-building-one-from-scratch-no-tf-keras-just-numpy. Accessed 17 Dec. 2023.