An Exploration into Neural Networks and their Applications

Ellen Kolesnikova

Table of Contents

Introduction

Artificial intelligence (AI) is used to perform increasing numbers of everyday tasks, including voice-to-text recognition, facial recognition, and more. One particularly convenient use case to me is my Kindle's handwriting recognition software, which transcribes my handwritten notes to text. As a coder, I was curious to understand how computers, which are only able to process a handful of very simple instructions, manage to accomplish such involved tasks. I researched this topic further and discovered the vast topic of neural networks and the math behind them.

In this paper I aim to describe the calculus behind neural networks and my process in creating my own digit transcription neural network in a formal yet accessible manner.

1.1 Neural Networks

Neural networks are machine learning models that mimic the processes of human brains in order to classify an input. For the sake of simplicity, let's focus on a classic example of a neural network problem - the transcription of handwritten digits. In a process called *training*, a neural network is given thousands of pieces of data and learns to distinguish between them. In our example, the neural network would be given a labeled dataset of images of handwritten digits, such as the MNIST digit dataset, and it would learn to classify the images as 0, 1, 2, …, or 9.

Neural networks are complex structures. They consist of *nodes* arranged in *layers*. Each node is connected to every node in its adjacent layers. A common way to visualize these networks is shown below — note the three key sections of a neural network: the *input layer*, the *hidden layers*, and the *output layer*.

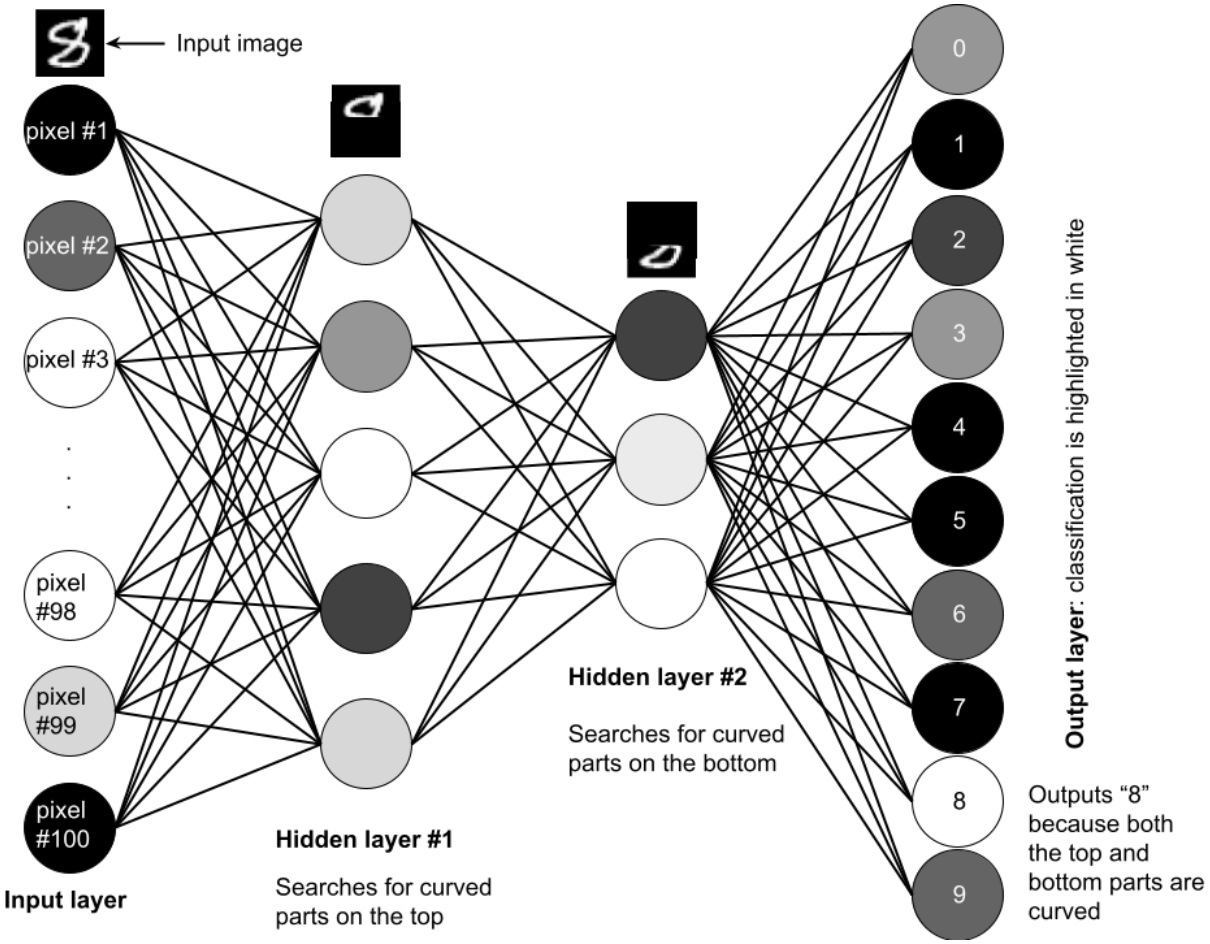


Figure 1

If the input data are images, as in our example, the images are separated into input nodes each containing one (numerical) pixel value based on the pixel's shade. For instance, if the input image has size 10 pixels x 10 pixels, the input layer of the neural network would consist of 100 nodes, each containing one pixel value. Each of these values are then used as inputs to the following layers of the neural network, as can be seen in the image above. The following layers, or the *hidden layers*, can be thought to represent specific distinguishing features of the images in the dataset. A neural network classifying handwritten digits could contain hidden layers that

identify the rounded parts of the digit. This would help it tell the difference between straight digits such as 1 and rounded digits such as 2.

An important thing to note here is that the neural network is not given these features to look for in the data. Rather, during training, the computer attempts to classify all data in the dataset. It keeps track of which inputs it misclassifies (for example, classifying an image of a 2 as a 1). Then, using calculus, the computer adjusts the features it's looking for, trying to minimize the errors it makes during classification (3Blue1Brown, 2017b).

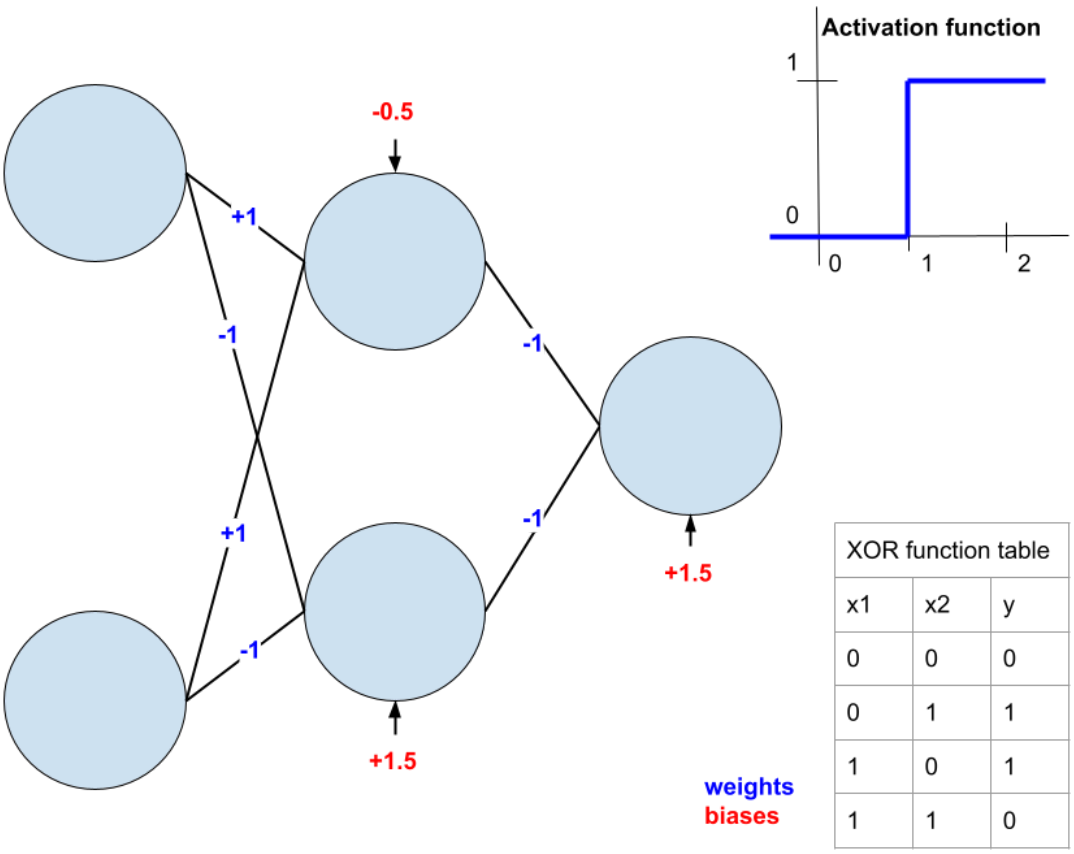
1.2 The Math in Neural Networks

Prior to this section, I gave a very high-level description of how neural networks work. In this section I will discuss what exactly computers are doing when creating and running a neural network.

Computers are only able to process numbers and operations on those numbers, and neural networks are no exception. Each layer performs mathematical operations (what I previously called “finding features”) on values of the previous layer to obtain the *activations* (real numbers from 0-1) of nodes in that layer. Activations can be thought of as “outputs” of nodes which are fed into nodes in subsequent layers. The activations of nodes in the penultimate layer then directly influence the final classification.

Let’s step away from abstract concepts and look into what operations are being performed on nodes in the neural network. To better illustrate this, let’s consider a simple example.

Imagine we have a neural network pre-trained to evaluate the XOR function (a boolean function defined in the image below) on two inputs. The neural network could look as follows:



XOR function table		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

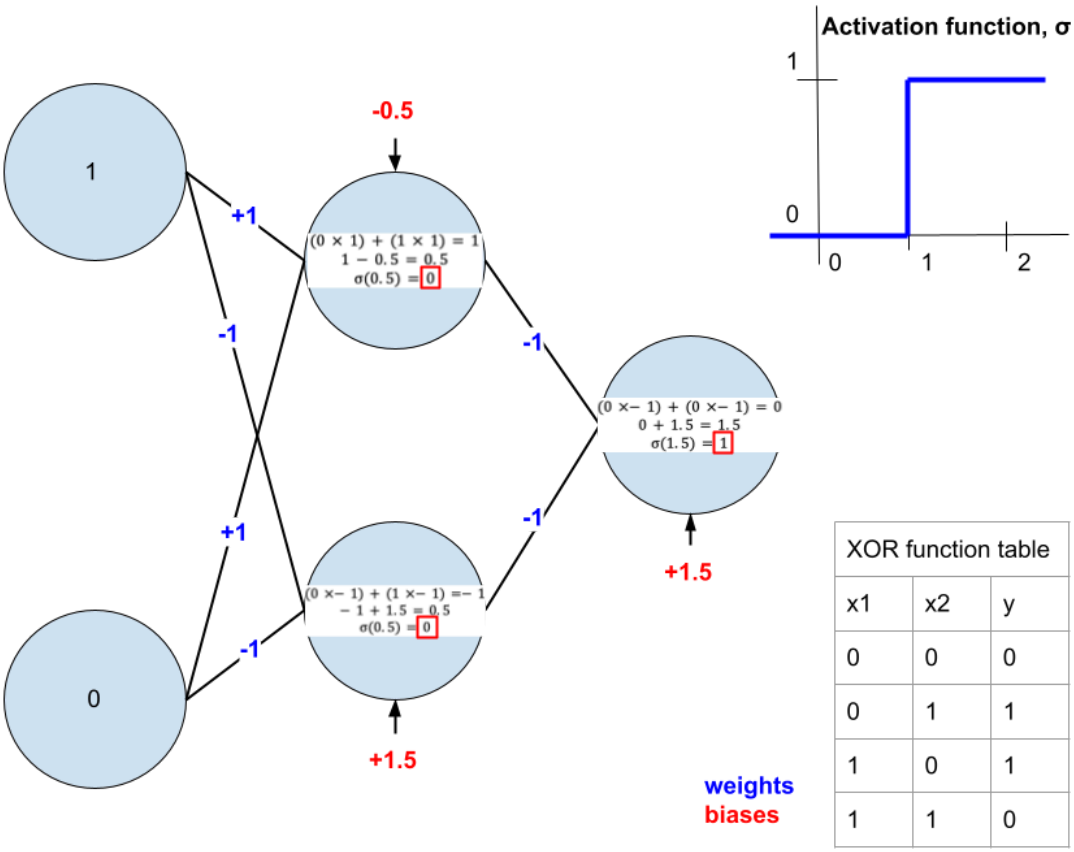
Figure 2

The main parameters that determine what a neural network does are *weights*, *biases*, and *activation functions*. Weights and biases are assigned by the network during training, while activation functions are assigned by the coder prior to training. Weights are given to all connecting edges of the neural network, highlighted in blue in the image above. These weights signify the importance of their corresponding edges. Biases, highlighted in red in the image, are given to all nodes in the network, excluding the nodes in the input layer. These biases offset the activations of each node. The neural network also has an activation function, such as the one shown above, which determines the final activation of each node.

The activation of each (non-input) node is given by the equation $a_l^n = \sigma(\sum_{i=0}^{k} (a_{l-1}^i \times w_{n_l}^{i_{l-1}}) + b_l^n)$, where the following is defined:

1) a_l^n is the activation of node number n in layer number l.
2) σ is the activation function of layer l.
3) k is the number of nodes in layer $l - 1$.
4) $w_{n_l}^{i_{l-1}}$ is the weight of the edge connecting n_l and i_{l-1}.

Let's go through an example to better illustrate this. Consider the XOR neural network from above. Plugging in 1 and 0 as the input values, we get the correct output: 1.



XOR function table		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 3

On the other hand, plugging in 1 and 1 as the input values gives us the desired output of 0.

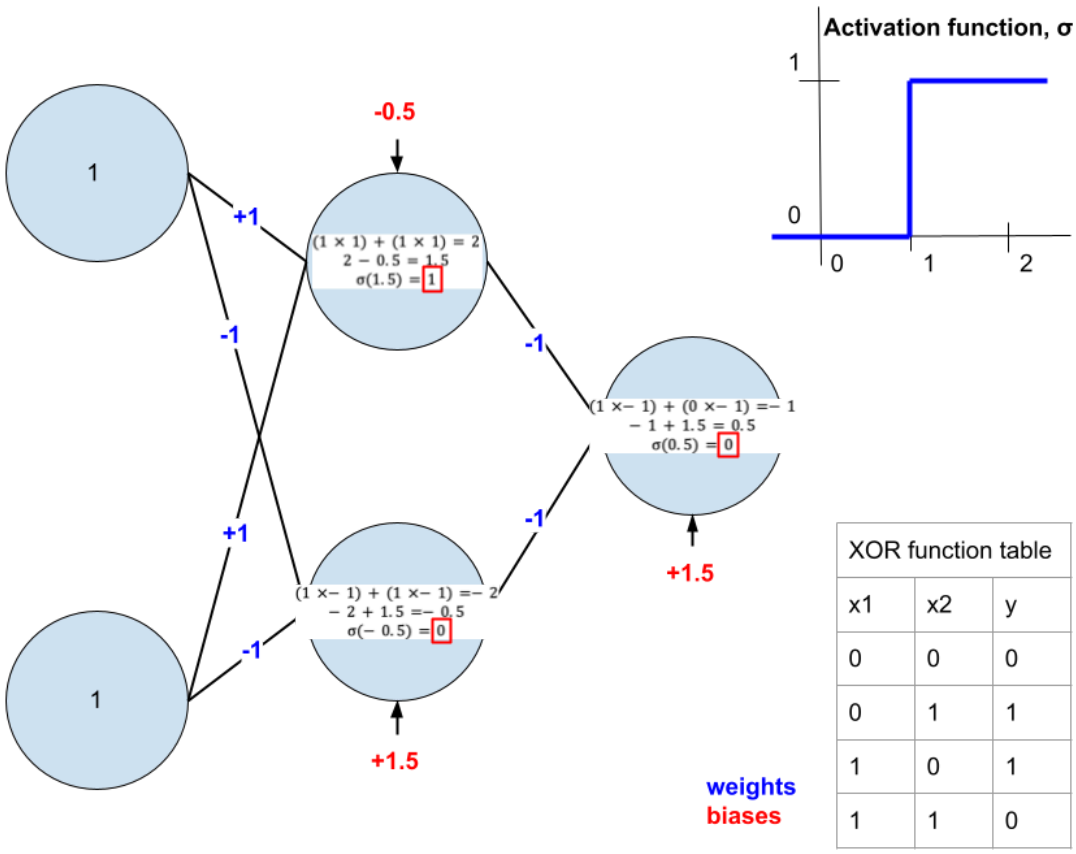


Figure 4

2.1 Introduction to Backpropagation

In this section I discuss how neural networks "learn", a process I previously mentioned very briefly.

Before a neural network begins learning, it assigns random weights and biases to edges and nodes. During learning, the neural network iterates through every element of the given

dataset. (This dataset is *labeled*, meaning that humans have created labels for each image by hand. For example, an image of a 2 would be labeled '2'.) It calculates the classification for each element (possibly incorrectly), and then finds the accuracy of its prediction through a *loss function*. This function, which is set by the coder, outputs a value that is higher when accuracy is low and lower when accuracy is high by comparing the expected and actual values for each classification (often done using the least-squares method). The neural network adjusts its weights and biases to decrease loss (improve accuracy) using an algorithm called *backpropagation*. It then repeats this process until the accuracy is high.

Let's take a look at how backpropagation works. Backpropagation is a recursive algorithm that finds a gradient of the loss function with respect to the weights and biases in the neural network. Using the negative of this gradient, we can adjust the weights and biases to minimize error in the neural network (3Blue1Brown, 2017a).

2.2 Backpropagation Calculus (Simplified)

Consider a simplified neural network with only one node in each layer (3Blue1Brown, 2017b). The steps of the backpropagation algorithm for this neural network are as follows:

1) Begin at the last layer of the neural network, L.
2) Consider every element i of the training dataset.
3) Calculate the partial derivative of the loss function of element i, C_i, with respect to the weight connected to the node in L, w^L. a^L is the activation of the node in layer L and b^L is the bias of the node in layer L.

a) Let's define $z^L = w^L \times a^{(L-1)} + b^L$. Then a^L, the activation of the node in L, is $\sigma(z^L)$, where σ is the activation function. Let's also define our loss function to be $(a^L - y)^2$, where y (a constant) is the desired activation.

b) The partial derivative, $\frac{\Delta C_i}{\Delta w^L}$, can be broken up into $\frac{\Delta z^L}{\Delta w^L} \times \frac{\Delta a^L}{\Delta z^L} \times \frac{\Delta C_i}{\Delta a^L}$ using the chain rule.

c) Taking these new partial derivatives, we get $\frac{\Delta z^L}{\Delta w^L} = a^{L-1}$, $\frac{\Delta a^L}{\Delta z^L} = \sigma'(z^L)$, and $\frac{\Delta C_i}{\Delta a^L} = 2(a^L - y)$. Thus, $\frac{\Delta C_i}{\Delta w^L} = a^{L-1} \times \sigma'(z^L) \times 2(a^L - y)$.

4) Take the average of all $\frac{\Delta C_i}{\Delta w^L}$ to get the partial derivative of the loss function *over all training examples* with respect to w^L: $\frac{\Delta C}{\Delta w^L} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\Delta C_k}{\Delta w^L}$.

5) Calculate the partial derivative of the loss function with respect to b^L, the bias of layer L, in a similar manner. $\frac{\Delta C}{\Delta b^L} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\Delta C_k}{\Delta b^L}$, where $\frac{\Delta C_k}{\Delta b^L} = 1 \times \sigma'(z^L) \times 2(a^L - y)$.

6) Calculate the partial derivative of the loss function with respect to a^{L-1}, the activation of the node in the previous layer.

a) $\frac{\Delta C_i}{\Delta a^{L-1}} = \frac{\Delta z^L}{\Delta a^{L-1}} \times \frac{\Delta a^L}{\Delta z^L} \times \frac{\Delta C_i}{\Delta a^L}$. Then $\frac{\Delta C}{\Delta a^{L-1}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\Delta C_k}{\Delta a^{L-1}}$

7) Step back a layer and calculate steps 3-6, substituting $\frac{\Delta C_i}{\Delta a^{L-1}}$ for $\frac{\Delta C_i}{\Delta a^L}$, and updating all other values to correspond to the new layer, $L - 1$. $\frac{\Delta C_i}{\Delta w^{L-1}} = \frac{\Delta z^{L-1}}{\Delta w^{L-1}} \times \frac{\Delta a^{L-1}}{\Delta z^{L-1}} \times \frac{\Delta C_i}{\Delta a^{L-1}}$ and

$\frac{\Delta C_i}{\Delta b^{L-1}} = \frac{\Delta z^{L-1}}{\Delta b^{L-1}} \times \frac{\Delta a^{L-1}}{\Delta z^{L-1}} \times \frac{\Delta C_i}{\Delta a^{L-1}}$. Repeat this process, going back more layers (hence the name *backpropagation*) until the input layer is reached.

8) Store all the partial derivatives of C, the loss function, with respect to each weight and bias to get the desired gradient.

We then adjust the weights and biases of the neural network using this gradient. The loss function is recalculated and this process is repeated until the accuracy sufficiently improves. For a relatively simple neural network, such as the digit recognition network from earlier, this backpropagation and adjustment process is repeated about 10 times (Manmayi, 2020).

Note that the above process is meant only for networks with one node per layer. However, this process extends naturally to cover more complex networks.

2.3 Optimization

In the backpropagation algorithm discussed in 2.2, we calculate the gradient of the loss function by iterating over each piece of training data. This is very computationally expensive, so one common optimization is to iterate over *mini-batches* of the training data instead. These mini-batches are randomly-chosen small subsets of the total training data. In every iteration of the backpropagation/adjustment process, a new mini-batch is chosen to create the new gradient. These mini-batch gradients are less accurate, but are much faster to compute.

Another common optimization is neural network pruning. This involves removing edges from the neural network which correspond to weights close to zero. In other words, we are simplifying the neural network structure by removing connections that do not have a significant impact on the output.

3.1 Coding a Neural Network

To apply my new knowledge in neural networks, I coded my own digit recognition algorithm, as discussed earlier in this paper (base code from Harvard's CS50 AI). This kind of neural network is very useful in everyday life — for example, as mentioned before, for the transcription of notes. My network does not transcribe letters yet, but that could be an improvement for the future.

My neural network is designed to classify a handwritten digit as one of 10 options: either 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. Therefore, my neural network was required to have 10 nodes in its output layer. Other than that, however, all choices about activation functions and numbers of nodes/layers were left to me to decide. I will discuss some decisions I made and their implications on the accuracy of my neural network.

3.2 Adjusting Parameters

Normally, in the coding of neural networks, an activation function is chosen for each layer of the network, rather than for each node. Usually, all hidden layers have the same activation function, while the output layer might have a different one. For this neural network, I decided to use the ReLU activation function for hidden layers and the softmax function for the output layer, a common decision in neural network programming (Saxena, 2021). These two functions are pictured below.

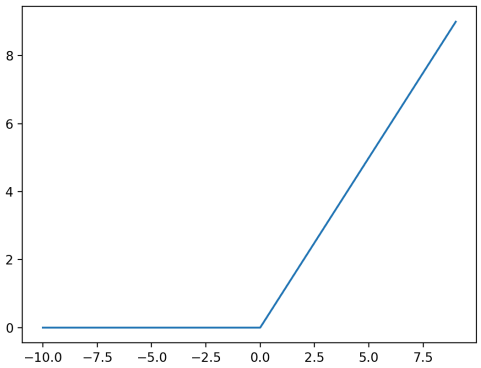


Figure 5 - ReLU

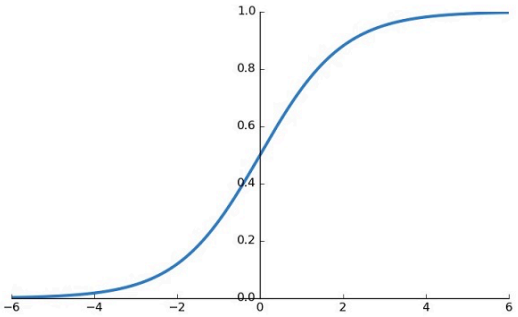


Figure 6 - Softmax

The ReLU function is often used as an activation function for hidden layers because of how quickly it can be computed — this speeds up training time considerably. Additionally, the ReLU function allows for less variation in the activation of nodes, since many nodes are given "0" activation. This prevents *overfitting*, which refers to the over-training of neural networks so that they classify training data accurately, but not real-life data.

I chose to use the softmax function as the activation function of the last layer because of its natural translation to probability, as it ensures that the sum of all output activations equals one. Hence, the activations of each output node can be interpreted as the probability that the specific node is the actual classification.

For layers and nodes, I found that having two layers with 64 nodes each was a good match for this problem. With too many nodes and layers, overfitting becomes likely. However, with too few, the neural network isn't able to accomplish complex tasks accurately. I experimented with different combinations of nodes and layers until I reached a good accuracy of 99%.

After deciding on these parameters, I ran my code to train my neural network using the backpropagation algorithm discussed earlier.

3.3 Neural Network Results

I trained my model on a random sample of half of the entire training database. Then I used the other half of the training database to test my model on previously-unseen data. I found that, using the parameters specified in Section 3.2, my model had a very high accuracy of 98.61%.

I used existing software to demonstrate the capabilities of my model, as shown in the images below.

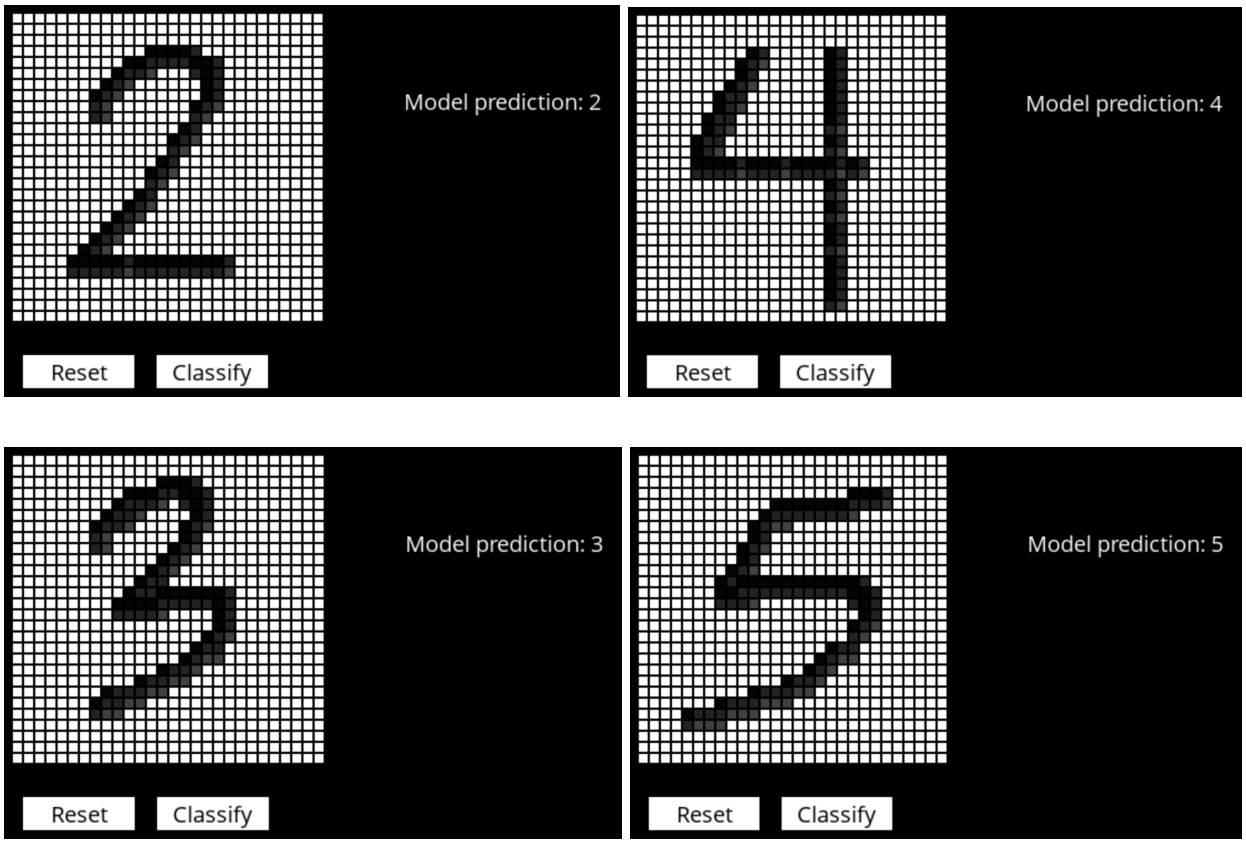


Figure 7

To make a more accurate model, I could have used more training data. However, this was not easily accessible — the MNIST dataset I used is the most thorough one freely available.

Additionally, I could have used larger mini-batches when training my network, which would have made the gradients found in the backpropagation algorithm more accurate.

However, neural networks aren't meant to magically interpret all kinds of data. There are some images of digits that are inherently unclear, and which neither humans nor my neural network can easily transcribe. These kinds of images contribute to the imperfect accuracy of my neural network.

Conclusion

Neural networks are extremely powerful tools which assist us in all aspects of daily life. This exploration into how they work under the hood and their limitations (some of which I discuss above in Section 3.3) helped me understand them much better. My new in-depth understanding will be crucial as I continue to study AI and implement and apply AI algorithms.

References

3Blue1Brown. (2017a). Backpropagation calculus | Deep learning, chapter 4. In *YouTube*. https://www.youtube.com/watch?v=tIeHLnjs5U8

3Blue1Brown. (2017b). What is backpropagation really doing? | Deep learning, chapter 3. In *YouTube*. https://www.youtube.com/watch?v=Ilg3gGewQ5U

Brownlee, J. (2019, April 20). *A Gentle Introduction to the Rectified Linear Unit (ReLU) for Deep Learning Neural Networks*. Machine Learning Mastery. https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

CS50's Introduction to Artificial Intelligence with Python. (n.d.). Cs50.Harvard.edu. Retrieved February 25, 2024, from https://cs50.harvard.edu/ai/2024/

Manmayi. (2020, June 6). *Choose optimal number of epochs to train a neural network in Keras*. GeeksforGeeks. https://www.geeksforgeeks.org/choose-optimal-number-of-epochs-to-train-a-neural-network-in-keras/

Refaeli, D. (n.d.). *Sigmoid, Softmax and their derivatives*. Themaverickmeerkat.com. https://themaverickmeerkat.com/2019-10-23-Softmax/

Saxena, S. (2021, April 5). *Introduction to Softmax for Neural Network*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/04/introduction-to-softmax-for-neural-network/#:~:text=Softmax%20is%20typically%20used%20in

Tanner, G. (n.d.). *Activation Functions*. Machine Learning Explained. https://ml-explained.com/blog/activation-functions-explained