# Building a Convolution Neural Network to Recognize Handwritten Digits From the Ground Up

Hwei-Shin Harriman

November, 2018

## 1 Introduction and Contextualization

The concept and application of convolutional neural networks (CNN) has reshaped the field of computer vision in recent years. Their ability to classify images has resulted in major advances in technology, with applications in self-driving cars, automatic tagging algorithms, image processing, photo search, obstacle detection, and more. But the process of training a CNN can seem like a simultaneously daunting and magical task. Given a set of training data that includes only well-tailored, hand-labeled images, a well-tuned CNN can develop its own representation of the data set and "teach itself" what it is seeing. Powerful packages like Tensorflow and Keras make it easy for anyone with some basic coding knowledge to run an example CNN and watch it "learn", while keeping all of the mathematical details behind the scenes. However, a CNN designed to recognize trees will likely have a different architecture than a CNN designed to recognize faces. So, copying and tweaking a network from a tutorial can be expected to be met with little success.

After having one such experience during my final project for Software Design last Spring, I decided to take this opportunity to dissect the math and architecture of convolutional neural nets by starting from scratch. The goal of this project is to code my own implementation of the popular "introductory" CNN using Python, no high-level packages and the MNIST database of handwritten digits. This will allow me to gain a deeper understanding of neural network terminology, as well as the relevant mathematical concepts like stochastic gradient descent, convolution, tensors, matrix operations, and computing concepts like backpropagation, pre-allocation, and debugging mindsets.

## 2 Key Terms for Convolutional Neural Networks

Before diving into the math of CNNs, it is necessary to establish context about the how neural networks learn, and define some fundamental terminology.

### 2.1 Overview of Network Training Process

All neural networks take a series of inputs and return an output. The input is fed through the network, one layer at a time, with each layer performing some computation on the output of the previous layer. For every input, the network makes a guess as to what the important information in the network is. Initially, this guess is pretty much guaranteed to be way off. So, the network compares its guess to the "target" output, or what the correct guess would have been. This is the overall network error. It then attempts to minimize this error function by calculating the gradient of the network error and taking a step in the direction of greatest descent. The size of this step is determined by the "learning rate", which is a small positive scalar that is multiplied by every component of the gradient vector. The learning rate is one of several "hyperparameters" that need to be tweaked in order to tune the efficiency of the network. Different network architectures have

different hyperparameters, and while many research papers suggest various ballpark values for them, the only thing that is known for certain is that their values should be picked based on whatever combination makes the network perform best. As more inputs are passed through the function, the overall network error should decrease and eventually converge. At this point, one would say that the network has been "trained" and is ready to move on to testing.

## 2.2 Overview of Network Architecture

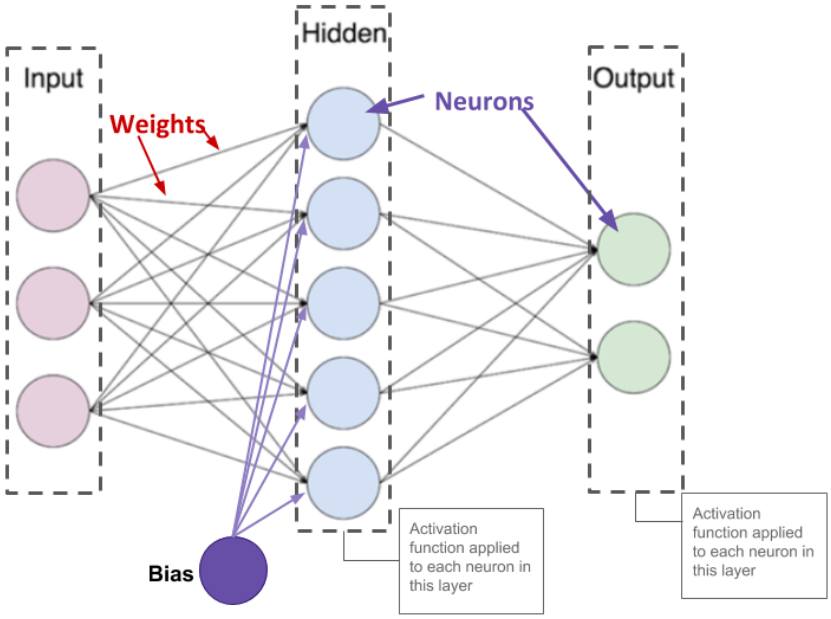



Figure 1: Diagram of a basic neural network. The neurons are connected by weights, and data is fed from left to right. The bias term is optional, and would influence each neuron in a layer in the form of a constant offset applied uniformly.

Neural networks are made up of a series of nodes and linkages. Typically, the nodes are referred to as neurons and the links are referred to as weights. The neurons are grouped in layers, where each layer has its own "activation function" that is applied to the inputted data.

The weights are scalar constants that are adjusted by the network as it trains. All of the weights in the network are stored in matrices or tensors[1], and the number of weights in the network directly corresponds to the number of components of the gradient vector that must be calculated for every training example. The process of training a neural network is therefore simply the process of adding and subtracting small differences to each weight in the network until the network's error is minimized.

There are also biases, which are another component that the network can adjust as it trains. There is one bias per layer, and each one is a constant offset that is added to the input of each neuron. These are often visualized as an additional neuron in each layer. I chose not to incorporate biases into the networks that I created for this project, because of the added complication of keeping track of them while coding.

## 2.3 Overview of Convolutional Neural Networks

Generally speaking, convolutional neural networks look for features, like outlines or edges, that help it to classify the data set it is training on. It does this by developing "filters," which are like low-resolution,

[1]In the context of neural networks, a tensor is simply an array of more than 3 dimensions.

zoomed in pieces of the training images. As an image is passed through the network, it is broken up and passed through multiple rounds of these filters. Each time it passes through a filter, the filter keeps track of how well it matches the piece of the image, and the network uses this information to guess which filters correspond to which output.

There are typically three different types of layers found in a convolutional neural network: convolutional layers, pooling layers, and a fully-connected layer. In a CNN, when a sample is passed through the network, it will always start by passing through a convolutional layer. This is usually a two-dimensional convolution, where a 2-dimensional filter (which is usually a 3x3 or 5x5 square matrix) examines small batches of the input at a time, and generates feature maps based on what information it thinks is important. Convolutional layers contain many of these filters[2] and each value of every filter can be adjusted through gradient descent.

Pooling layers are used to downsize the data further. It is common for pooling layers to examine a small patch of pixels and take only the largest value in that square. This is otherwise known as max pooling. They are always applied after a convolutional layer.

At the end of the CNN, before returning the network's guess, there is usually a fully-connected layer. Fully-connected means that every neuron from the previous layer is connected to every neuron in the next layer. Roughly speaking, this functions as a type of voting system for the network. Each filter in the previous layer "votes" on how much it thinks the image that was passed in corresponds to one of the possible outputs, or guesses. The output node that receives the highest value will be the network's guess.

# 3 Mathematical Background

Convolutional neural networks are, in their essence, extremely high-dimensional functions. These functions contain local minima and maxima, and can therefore be minimized through gradient descent. Due to the sheer number of components involved in the gradient vector of a neural network (one component per weight, or link in the network), it is unrealistic to calculate each component individually. Instead, we apply the multivariable chain rule which states that partial derivatives of a multi-dimensional function can be calculated by taking the individual partial derivatives of each applicable variable and chaining them together. For instance, say we wanted to solve for the output of a multi-dimensional function $E$ in terms of a single component $w$. If we know that $E$ is directly affected by $a$, and we know that $w$ directly influences $a$, then we could write:

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial a}\frac{\partial a}{\partial w} \tag{1}$$

This definition can be extended to any number of variables, and it is the core concept behind the back-propagation algorithm. Backpropagation greatly decreases the computational power needed to calculate the gradient of a neural network. It is shorthand for the "backwards propagation of errors," since the error is calculated at the output of the network and fed backwards through the network[3]. Knowing how to implement this algorithm is central to understanding how neural networks learn, so the following sections will break down the derivation of this algorithm.

## 3.1 Backpropagation

### 3.1.1 Sigmoid Function

According to Wikipedia, sigmoids are mathematical functions that have a characteristic "s"-shaped curve. It is very common to see sigmoid functions used as activation functions in neural networks.

[2]The number of filters can be anything, but multiples of 2 are common, such as 8, 16, 32, 64. This is due to the computational efficiency of powers of 2.

[3]source: Wikipedia

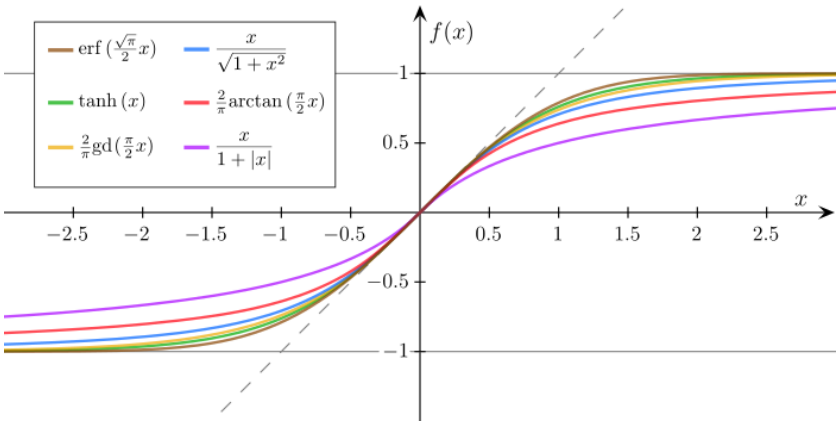



Figure 2: Graph of various sigmoid functions. Source.

You can clearly see the "s"-shaped curve present in the sigmoid functions in Figure 2.

One of the most common sigmoid functions is the logistic function, given by the equation,

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

One of the reasons why sigmoid functions are such a popular choice for neural networks is because their derivatives are simple and computationally efficient. It turns out that the derivative of Equation (2) can be expressed by,

$$\sigma'(x) = s(x) \cdot [1 - s(x)] \tag{3}$$

where $\sigma(x)$ is the sigmoid function[4]. This version of the derivative is more efficient from an implementation standpoint. This is because the derivative is made up of simple multiplication and subtraction operations, involving the output of the sigmoid function itself. This equation will be applied to our dummy network in the form of a "nonlinear activation function," which will be defined shortly.

### 3.1.2 Notation

Backpropagation involves a lot of variables, and it quickly becomes challenging to keep track of all of it. Due to this observation, I have included a table that contains all of the notation that will be included in the following section.

| Variable | Description |
|---|---|
| $a_j$ | Weighted sum of inputs to hidden node $j$ |
| $w_{ji}$ | Weight connecting the $i$th node of the input layer to the $j$th node of the output layer |
| $\sigma(.)$ | Logistic sigmoid function |
| $z_j$ | Output of the $j$th hidden node |
| $y_k$ | $k$th node of the output layer |
| $t_k$ | Target of the $k$th node of the output layer |
| $\alpha$ | Learning rate of the network |

Table 1: Table of general notation that will be used in the derivation of backpropagation.

[4]If you are interested in seeing the proof, see here.

### 3.2 Backpropagation Derivation

To make the important information of backpropagation more accessible we will go over the implementation using a simplified, dummy network. This network is made up of one input layer, one hidden layer, and one output layer, and all of the layers will be fully-connected. The network architecture is shown in Figure 3. The derivation we are about to go over comes from Chapter five of this textbook. For a more detailed breakdown of the backpropagation derivation, see this video.

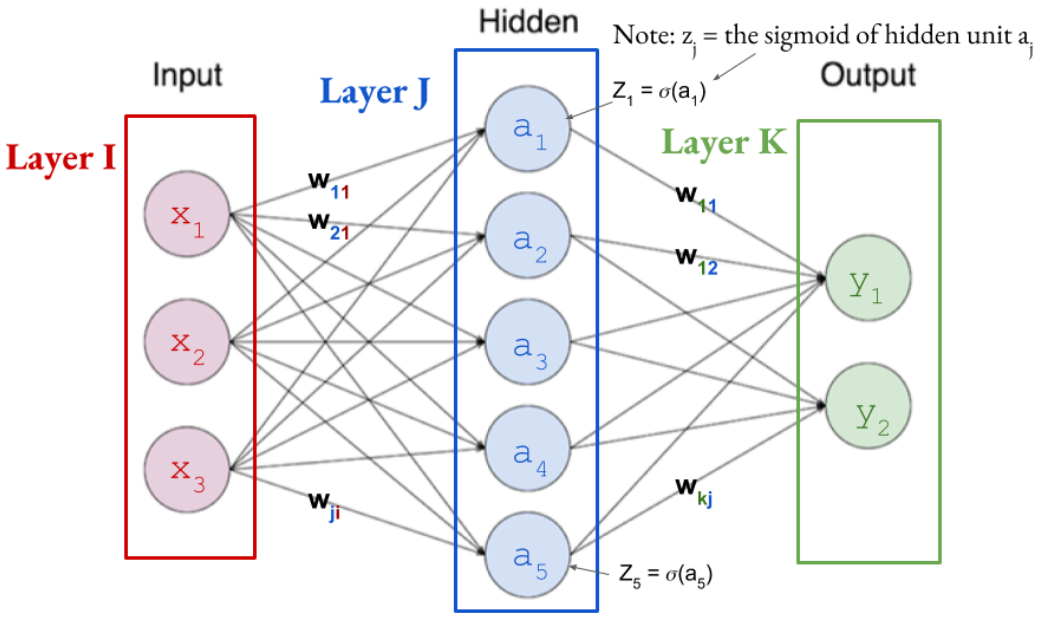



Figure 3: This figure provides a visual guide to the notation that will be used in the derivation for backpropagation. Subscripts $i$, $j$, and $k$ will be used to denote that the variable in question belongs to the input, hidden, or output layer respectively. Note that $z_j$ refers to the output of the hidden node $a_j$, and is simply the value of $a_j$ passed through a sigmoid function.

Each unit computes the weighted sum of its inputs of the form,

$$a_j = \sum_i w_{ji} z_i \tag{4}$$

where $z_i$ is the "output" of the previous node (which is just the input layer), and $w_{ji}$ is the weight associated with that connection. Notice that the definition of $a_j$ in Equation (4) is the same as the dot product.

For example, the value of $a_1$ from Equation 3 can be computed as

$$a_1 = w_{11} z_1 + w_{12} z_2 + w_{13} z_3 = \sum_i^3 w_{1i} z_i = w_1 \cdot z \tag{5}$$

The sum (4) is then transformed by the nonlinear activation function (sigmoid), which gives the activation $z_j$ of unit $j$ as

$$z_j = \sigma(a_j) \tag{6}$$

Now, consider the error of the neural network to be the sum-of-square error, calculated as follows:

$$E = \frac{1}{2}\sum_{k \epsilon K}(y_k - t_k)^2 \tag{7}$$

where $E$ is the error of a single training iteration. $y_k$ is the network output, or guess, and $t_k$ is the expected output. The goal of backpropagation is to minimize this error function. To do so, we must calculate the rate of change with respect to each connective weight. Using the multivariable chain rule, we can find the derivative of $E$ with respect to $w_{ji}$,

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} \tag{8}$$

Notice that the change of $a_j$ with respect to the weight $w_{ji}$ is the same as $z_i$:

$$\frac{\partial a_j}{\partial w_{ji}} = x_i \tag{9}$$

Now we redefine the change in error, $E$ with respect to the activation unit $a_j$ to be $\delta$, otherwise stated as,

$$\delta_j = \frac{\partial E}{\partial a_j} \tag{10}$$

Therefore, Equation (8) can be expressed as,

$$\frac{\partial E}{\partial w_{ji}} = \delta_j z_i \tag{11}$$

This equation defines the components that make up the gradient vector used to optimize the network. In order to evaluate the partial derivatives, we begin by calculating the value of $\delta_j$ for each hidden and output unit in the network, then apply Equation (11). The values of $\delta_j$ cannot be calculated simultaneously due to missing pieces of information, so instead we begin with only the $\delta_k$ values in the output layer, and then backpropagate through the network to fill in the rest of the values.

For output units, the value of $\delta_k$ is just,

$$\delta_k = y_k - t_k \tag{12}$$

where $y_k$ is the output of the network at the $k$th node, and $t_k$ is the desired target output at that node.

To calculate the $\delta$'s for hidden units, again make use of the chain rule:

$$\delta_j = \frac{\partial E}{\partial a_j} = \sum_k \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial a_j} \tag{13}$$

Substituting the definition of $\delta_k$ back in, we obtain the backpropagation formula,

$$\delta_j = \sigma'(a_j)\sum_k w_{kj}\delta_k \tag{14}$$

By this definition, we are able to obtain the value of $\delta$ for any particular hidden unit by propagating the $\delta$'s backwards from units higher up in the network, as shown in Figure 4, below. The backpropagation formula can be recursively applied to evaluate all $\delta$ values regardless of the network architecture because of this property.

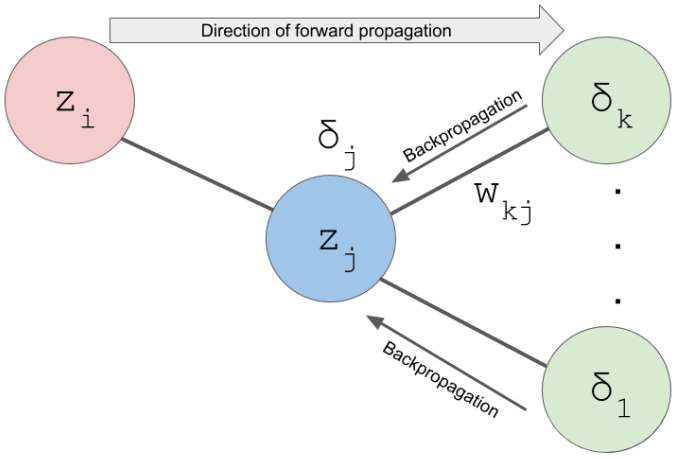



Figure 4: This figure demonstrates the flow of information in the forward and backwards passes of the network. Upon completing the forward pass, the $\delta_k$ values are calculated, and then used to recursively calculate all of the $\delta_j$ values in the network. These values are then used to calculate the gradient of the error function of the network.

Once all of the $\delta$ values have been calculated, there is enough information to calculate the derivative of the error function (8). All that is left to do is subtract each component of the gradient times some learning rate $\alpha$ (a small, positive value whose only constraint is that it must make the network converge) from each corresponding component of the weights matrix,

$$W_n = W_o - \alpha \bigtriangledown E \tag{15}$$

where $W_n$ is the new matrix of weights, $W_o$ is the old matrix of weights, and $\bigtriangledown E$ is the matrix of gradient components.

### 3.2.1 Error Backpropagation Process

To summarize, the steps for performing backpropagation for a neural network with logistic sigmoid activation units and a sum-of-squares error function are as follows:

- Apply an input vector $x_n$ to the network and propagate forward using Equations (4) and (6) to find the activations of all hidden and output units.
- Evaluate the $\delta_k$ for all the output units using Equation (12).
- Backpropagate through the network using the $\delta_k$'s from Equation (14) to obtain the $\delta_j$ for each hidden unit in the network.
- Use Equation (11) to calculate each component of the gradient.
- Update the weights matrix by subtracting each component of the gradient times the learning rate using Equation (15).

These steps can be generalized to any neural network, no matter the activation function or the error function. For instance:

- Every step of training the neural network begins with feeding an input vector (otherwise thought of as a single, high-dimensional point) all the way through the network. The output will be the network's guess, based on the current values of the weights.

- Calculate the difference between the network's guess and the target output.
- Use this difference to backpropagate through the network, using the concept of the multivariable chain rule to solve for the error with respect to the weights of each layer, $\frac{E}{w_{ji}}$. This gives the components of the gradient vector at this high-dimensional point (the input vector).
- Update the weights by subtracting each component of the gradient vector times the learning rate using Equation (15).

### 3.3 Validation with Numerical Differentiation

Because of the high-dimensionality of neural networks, it can be rather intimidating to sanity check the coded implementation of backpropagation despite its obvious importance. The most common way to check if backpropagation has been correctly implemented is by applying numerical differentiation. This can be accomplished by calculating derivative with finite differences. For some small value $\epsilon$ (on the order of $10^{-9}$),

$$\frac{\partial E}{\partial w_{ji}} = \frac{E(w_{ji} + \epsilon) - E(w_{ji} - \epsilon)}{2\epsilon} \tag{16}$$

should be calculated for each component of the weights matrix and compared to each component of the gradient obtained through backpropagation. The difference between the two should be on the same order as $\epsilon$, if not smaller. For instance, after implementing my backpropagation algorithm, I found that differences were on the order of $10^{-11}$.

### 3.4 Convolution

Due to the number of topics that will be covered in this paper, I will not go into the derivation of two-dimensional convolution. Instead, I have provided links to explanations that I found to be most helpful in my research for this project. I will be discussing the implementation of these concepts in more detail in Section 4.3.

- For a simple implementation of 2D convolution using Python and popular filters, see here.
- For one of the more intuitive explanations of the derivation of backpropagation for convolutional layers, see here. The basic idea is that both forward and backwards propagation through a convolutional layer involves convolution.
- For general information about the process of setting up a convolutional neural network, see this Stanford course assignment.

### 3.5 Softmax and Cross Entropy

In the Quantitative Calculation section you will see that in the process of implementing a feed-forward neural network to train on MNIST I switched over from a sum-of-square error function to a cross entropy error function. A cross entropy error function is almost always paired with a softmax activation function on the output layer of the network. This is due to the fact that this combination of error and activation function allows the network to train based on "likelihood." The softmax activation function is given as,

$$y_k = \frac{e^{z_j}}{\sum_c^{n_c} e^{a_c}} \tag{17}$$

where $n_c$ is the number of output classes, $z_j$ is again the output of the last hidden layer and $a_c$ is the input to the output layer. We can apply the value of $y_k$ to the cross entropy error function for a multi-class output, which is given as,

$$E = -\sum_{k}^{n_c} t_k \log(y_k) \tag{18}$$

where $n_c$ is the number of possible classes, and $t_k$ is a one-hot target vector (one-hot meaning that all of its entries are zero except for one index which corresponds to the correct answer). This combination of activation and error function are well-suited for neural networks where the goal of training is classification. This is due to the fact that the output off the prediction will always be between 0 and 1, and can thus be interpreted as a probability. Since there are 10 possible classifications for hand-written digits (0 through 9), using this type of neural network greatly improves the robustness of the network. You can find a full derivation of the backpropagation algorithm for a softmax/cross entropy network here.

# 4 Quantitative Calculation

The process of building a neural network using only Numpy functions requires many steps. Because the convolutional neural network implementation is extremely complex to manage, I began by building a robust, yet slightly simpler network to use as a point of comparison. I will begin by talking about the development of this network, the process of which taught me a lot a about neural networks in general, and then I will discuss the full convolutional neural network approach.

## 4.1 Data

To train and test my network, I decided to use the MNIST data set of hand-written numbers. I chose this particular data set because it is an extremely popular, well-defined and pre-processed data set that is especially useful for education purposes. This is most likely due to the fact that the images are all 28x28 pixels, the numbers are all roughly the same size, and it is not necessary to have an overly complicated neural network to obtain a fairly high accuracy with it. Also, since I would be building my networks without helper functions from popular neural network libraries and packages, using a data set that many others have already used made it easier to seek out process and debugging information online.

Using online resources, I was able to download the MNIST data set in a .csv format. The data set contains four important pieces of data: 60000 training images, 60000 corresponding training labels, 10000 test images, and 10000 test labels. The data was originally structured with the first index of each number corresponding to the target, and the next 784 indices corresponding to each pixel in the 28 by 28 image, such as the ones shown in Figure 5. To prepare the data for training and testing, I separated the target values from the pixel values to be stored in 4 separate vectors (training set, training labels/target values, test set, test labels).

The original values contained in the vectors representing the images were on a scale of 0-255, and the target values were 0-9. Knowing that the output layer of the network would be 10 neurons, each with a range of possible values from 0-1 (with 1 corresponding to the network being 100% certain that it was guessing correctly), I turned the training labels into 10x1 vectors containing all zeros except for the index corresponding to the correct digit. For instance, if the correct number was 4, the corresponding target vector was,

$$[0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 0,\ 0,\ 0,\ 0]$$

Additionally, I rescaled the intensity values of the images, so that they were also on a scale of 0-1. To begin, I would pass one image through the network at a time, but eventually I was passing the entire data set through the network in batches, multiple times. This led to the need to randomly shuffle the order of the

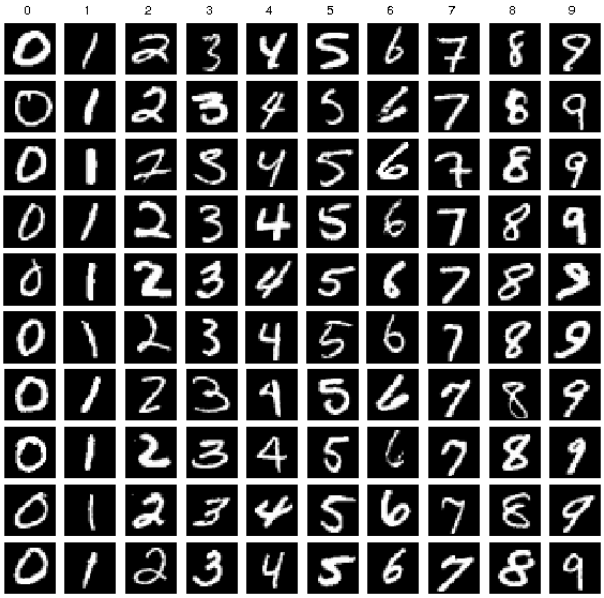



Figure 5: Some examples of handwritten numbers that are a part of the MNIST data set. Each image has a resolution of 28x28 pixels. When feeding the images through the network, the image is reshaped into a row vector of dimensions 784x1. Source.

training set while maintaining the relationship between the images and the labels, as well as splitting the data set into subsections, otherwise known as batches. I will talk more about batches in my implementation of stochastic gradient descent in Section 4.2.5.

## 4.2 Feed-Forward Network Implementation

The first network I constructed contained one input layer, two fully-connected layers (every neuron in the previous layer is connected to every neuron in the following layer), and one output layer. The input layer had 784 neurons, one for each pixel in the input image, the fully-connected layers had 625 neurons each, and the output layer had 10 neurons, one for each possible guess. Weights were connected between each layer, giving me three sets of weights matrices with dimensions 784x625, 625x625, and 625x10 respectively. At the beginning of training, the weights were randomly initialized within a standard normal distribution of mean 0 and variance 1. A visual of this network architecture is shown in Figure 6, below.

### 4.2.1 Forward Pass

To begin a training iteration, a single 784x1 flattened image vector was fed into the input layer of the network. Each of its 784 values was connected to every neuron in the first hidden layer, so I took a dot product to obtain the $a_j$ input value for every node in the layer, by Equation (4). Both of the hidden layers had logistic sigmoid activation functions like the one shown in Equation (2). Feeding the $a_j$ values through the sigmoid gave me the $z_j$ value of every neuron in the first hidden layer. These values then became the input for the second hidden layer, and the same steps were repeated.

Between the second hidden and output layer the same dot product operation was used to calculate the $a_k$ values, but instead of using a sigmoid function these values were fed through the softmax activation function from Equation (17). At this point, the output of the network has been obtained. I subtracted each of the target values (either 0 or 1) from the network "guess" values (float decimals between 0 and 1) to obtain the $\delta_k$ value from Equation (12), and began backpropagation.

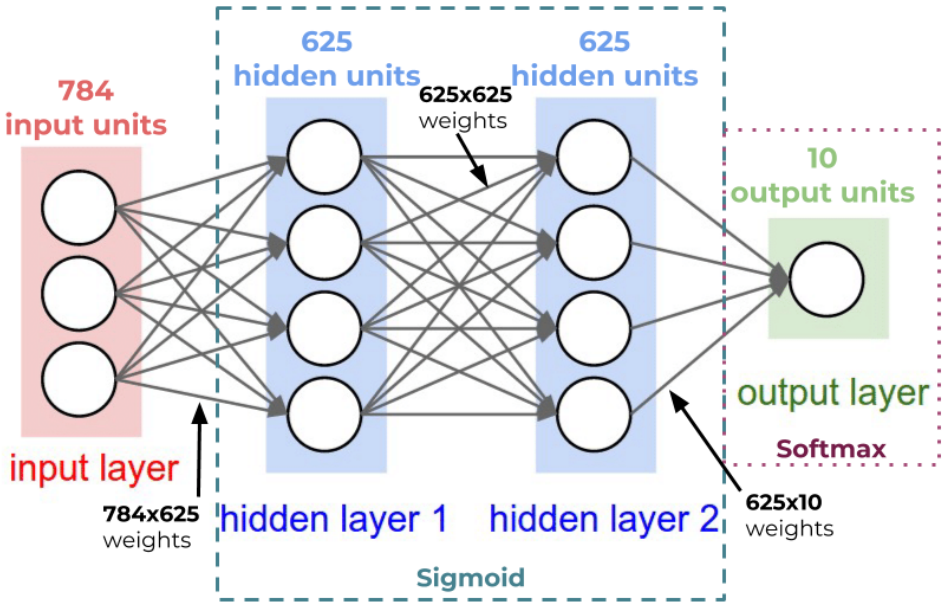



Figure 6: This figure shows the network architecture implemented in the feed-forward implementation. The dotted boxes group layers by their activation functions (either a logistic sigmoid or a softmax). The error function for the entire network is the cross-entropy error function. All of the layers in the network are fully connected, so every unit in a layer is connected to every unit in the previous and following layers. Original image source.

#### 4.2.2 Backpropagation and Updating the Weights

Using the $\delta_k$ value from forward propagation, I then implemented Equation (14) to obtain the values for $d_j$ for all of the remaining 1250 neurons. In practice, this involved performing a matrix multiplication between the deltas obtained in the previous layer (starting with $\delta_k$). and the weights matrix connecting the current layer to the previous layer[5]. The resulting matrix was then multiplied elementwise by the stored outputs of the current layer[6], $z_j$, and fed through the derivative of the sigmoid function from Equation (3). This process was then repeated for the preceding layers until all $\delta_j$ had been obtained. Note that for the last round of backpropagation between the first hidden layer and the input layer, there is no activation function for the input layer, so the $a_j$ values are simply $x_i$, where $x_i$ is the original input value at that neuron.

To calculate the gradient, I began with a list of pre-allocated matrices of zeros, whose dimensions matched those of the current weights. Then I iterated forwards through each layer in the network, performing matrix multiplication between the $\delta$'s of the next layer and the $a_j$ values of the current layer to obtain a new new matrix of gradient components[7].

To update the weights I multiplied the gradient components by the learning rate (in my tests, I found that a learning rate of .001 gave the best results) and subtracted it from the current weights matrix.

#### 4.2.3 Training and Testing

The process outlined above makes up one iteration of training. Upon completing an iteration, I stored the cross-entropy error that was calculated when the network guessed. This became the "loss" of the network. I

[5]for the second hidden layer this was the 625x10 matrix.
[6]Beginning with the second hidden layer.
[7]for the first matrix this resulted in a matrix of dimensions 784x625 gradient components. For the second layer this was 625x625 and the third layer it was 625x10.

also counted every time the index of the network's maximum output value (the number it is "most certain" the input image corresponds to) was the same as the target output. At the end of each iteration this was returned as the accuracy. Over hundreds of iterations the network loss decreased and the accuracy increased, which is evidence of the network "learning."

However, a network that has achieved 100% accuracy on its training set still might not do well on a subset of images that was not included in training. This is where the test set becomes important. A well-trained neural network will be able to generalize the data that it was trained on, and accurately apply that information to similar data that was not in the training set. So, it becomes important to also periodically pause training and feed the entire test set through the network, without updating the weights, so as to obtain data points on the average test set loss and accuracy over time.

#### 4.2.4 Validation of Backpropagation Implementation

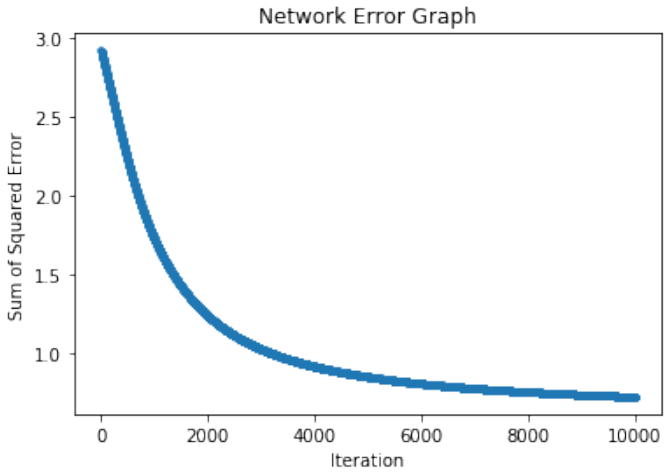


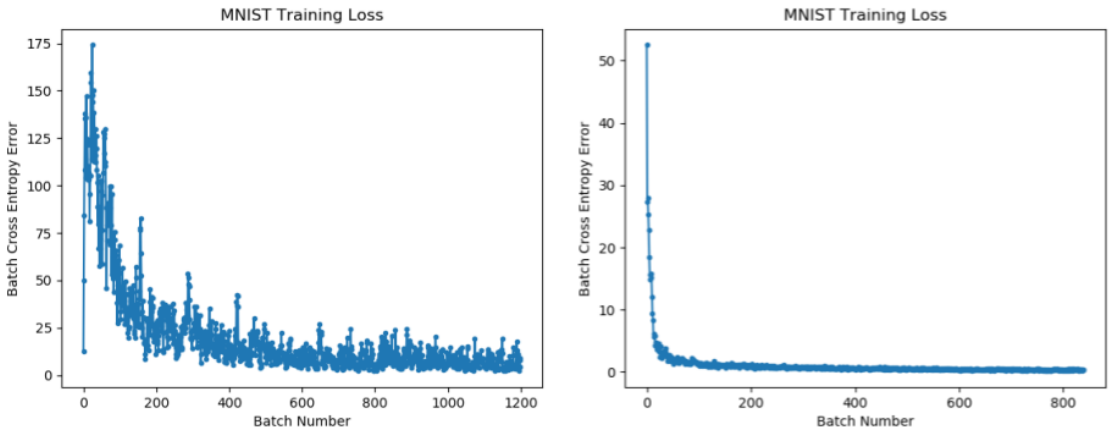

Figure 7: This figure demonstrates the functionality of my original backpropagation algorithm, using a sum-of-square error function. Here, the input and target vector were equal, and the weights were randomly initialized at the beginning of training. The network trained to output the same vector it was given.

Another issue that needs to be considered when testing neural networks is that the implementation of backpropagation is correct. In addition to using the concept of numerical differentiation to test each component of the gradient vector, another simple trick is to feed a neural network one (high-dimensional) point repeatedly, and see if the loss quickly approaches zero. Any neural network that has a proper implementation of backpropagation will be able to learn to reproduce individual points. Figure 7 shows an example of the loss of my network as it learned to reproduce one image given randomly initialized weights. Note that this example made use of a sum-of-square error function as opposed to the cross-entropy error function, but the application is the same.

#### 4.2.5 Stochastic Gradient Descent, Batches, and Epochs

One issue that arises in training is that a network will try to overfit to a data set. This can result in loss graphs that look jagged and very messy, or in the network getting stuck in a local minimum. This can cause the loss to stagnate at an undesirable value, which in turn results in a lower accuracy or higher loss on the test set. To counter this, I implemented a variation of iterative gradient descent, called stochastic gradient descent. Instead of updating the weights after every training sample, I split the data set up into batches, passed each image in a batch forward and backward through the network to calculate its gradient, summed up all of the gradients from the current batch, and subtracted the resulting "combined gradient" from the weights matrix. Though this may perhaps seem counter-intuitive and it can in fact lead to bigger,

less accurate steps, if a good relationship between the batch size and learning rate is selected, then stochastic gradient descent prevents the network from trying to overfit to each training example and instead take a broader, more generalized perspective. A well-tuned neural network should be able to start with a random point and random weights and quickly converge to a near-zero loss. Therefore, choosing the batch size is actually quite important, and is also considered a hyperparameter that must be picked based on testing. In my training I found a batch size of 500 images to give the best results. Figure 8 shows cases of overfitting and optimal fitting.




Overfitting: Too much noise, likely the learning rate was too high and the batch size was too small.

Good Fit: Quickly converging at zero loss, small and precise steps with little noise.

Figure 8: These examples from training my network show cases of overfitting as well as optimal fitting. The overfitting is jagged and very noisy due to each step of the gradient being too large, while the optimal fit is much smoother and more precise. The batch size and learning rate of the figure on the left were 250 and .05 respectively, while the ones on the right were 500 and .001.

The last factor to consider is "how many times do I want my network to see each training example?" This is the number of epochs. Before each subsequent epoch, I shuffled the training set so that the network would not be able to learn the order of the images it was seeing, which further encourages it to focus on general patterns as opposed to unimportant ones. I found that 5 epochs provided enough iterations for the network to converge, while 7 squeezed out barely one percent more accuracy in both the test set and the training set. Loss and accuracy graphs for training and testing of this network can be found in Section 5.

### 4.3 Convolutional Neural Network Implementation

While the basic ideas that have been covered in the paper thus far are still applicable to convolutional neural networks, adding the various layers that are paired with them is a fairly significant step up in complexity. The network architecture I selected was based on this introductory Tensorflow tutorial. However, I scaled down the number of filters to help combat some bugs I encountered along the way. My version of the architecture contains an input layer, followed by a convolutional layer with a ReLU[8] activation function, followed by a max pooling layer, then another round of convolutional/ReLU and pooling, followed by the output layer[9].

[8]This is another nonlinear activation function, like a sigmoid. It is defined as $\max(0, x)$. I implemented a "leaky" ReLU function, which allows small negative values to pass. For more about ReLU's, see this source.

[9]My original architecture also included a dropout layer and an additional fully-connected layer just before the output layer, but those were eventually cut in an attempt to simplify my process.

The network architecture is visualized in Figure 9.

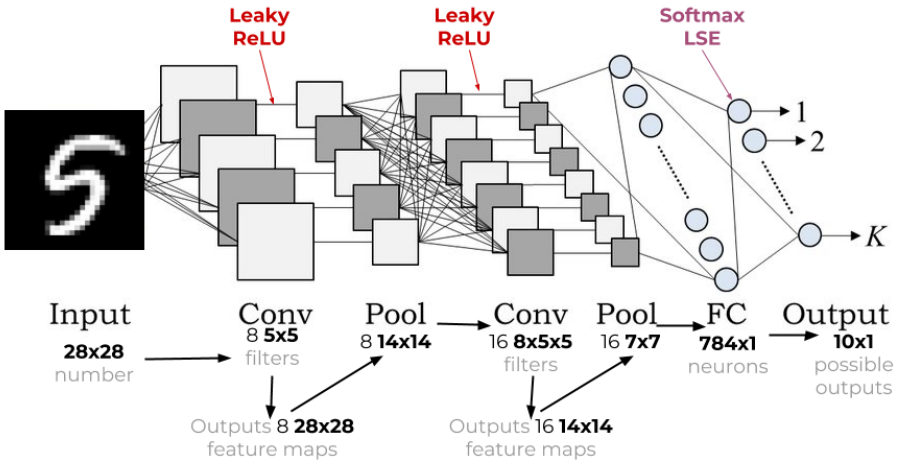



Figure 9: A visualization of the network architecture that I implemented using convolutions. The relevant dimensions as well as forward flow of data through the network is shown beneath each subsequent layer. The points at which nonlinear activation functions are applied on the forward pass are also indicated. Original image source.

### 4.3.1 Forward Pass

To implement the forward pass, the input image must first be reshaped into its 28x28 form. The first convolutional layer takes 8 5x5 filters with randomly initialized values and performs 2D convolution on the image. The output of this layer is then 8 28x28 "feature maps," where values close to 1 correspond to pixels in the input image that were "strongly activated" by the filter, and values close to zero correspond to pixels that were not[10]. After convolving, I applied a leaky ReLU activation function to each of the generated feature maps, so that all values that were less than zero were scaled down significantly, and all values greater than zero were left untouched.

The max pooling layer was implemented with a window size of 2x2 and a stride of 2. This means that there was no overlap as the pooling window iterated over each of the feature maps. It also conveniently halves the dimensions of the feature maps, such that the output of the max pooling layer was 8 14x14 feature maps, where only the maximum value of each iteration was passed on. A visual representation of a pooling layer is shown in Figure 10.

In order to perform the second convolution, the 8 14x14 feature maps needed to be reshaped into one "image" that could be passed into the second convolutional layer. One way to think about this is by stacking all of the 14x14 filters on top of each other, creating a single image with 8 channels, similar to how a color image has 3 channels, corresponding to red, green and blue. In order to perform convolution on this image, the 16 filters must also be three-dimensional: 8x5x5. This allows the filter to capture information through all of the channels at once. The output of this layer is then 16 14x14 feature maps[11]. Using this technique allows the network to search for both high-level and low-level features of the images. Examples of high-level features would be the circle present in the numbers zero, six, eight, and nine, or the line present in numbers one, four, seven, and nine. Examples of low-level features would be lines that run diagonally top-right to bottom

[10] In order to get the output of the 2D convolution to have the same dimensions as the input, I zero-padded the input image. For more information on zero-padding, see here.

[11] Zero padding was employed again.

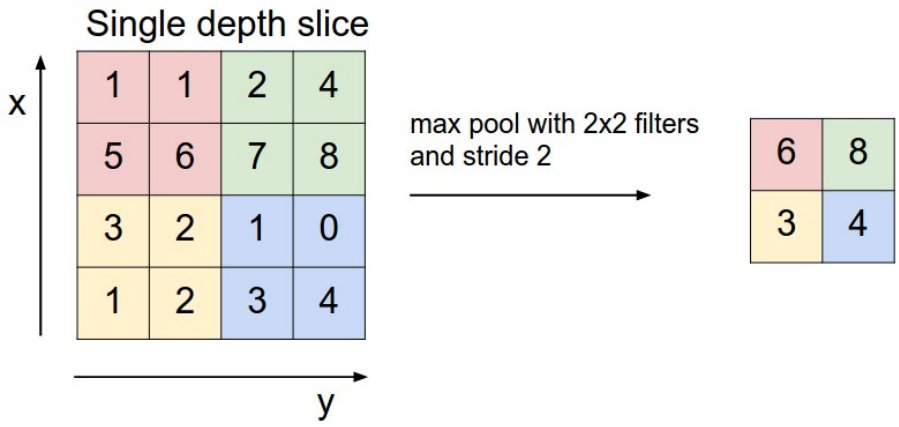



Figure 10: This figure demonstrates the concept of a max pooling layer. I also implemented a pooling layer with the same characteristics as described in the image. The window size of 2x2 and a stride of 2 is common because it scales down the input image by half. Source.

left, or a quarter-circle oriented along the bottom axis. Figure 11 shows some examples of high-level versus low-level features in a data set of faces.

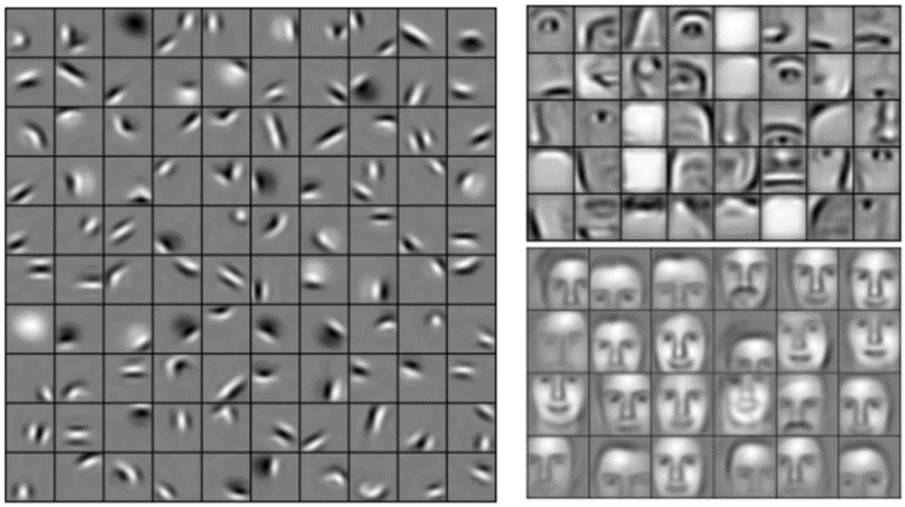


Figure 11: Some high-level features are shown in the upper and lower right-hand corners. These features are easily distinguishable by humans, and are clearly face-like. Low level-features are shown in the square of images on the left, where bright patches that might correspond to foreheads, chins, or the bridges of a nose are being differentiated. These feature maps are harder to decipher, but they provide important information for the network to learn. Source.

After convolving, the feature maps are again fed through a leaky ReLU activation function and pooled, bringing the dimensions down to 16 7x7 activation maps. At this point, the activation maps are chained together and stretched into a one-dimensional vector of dimensions 784x1, and fully-connected to the output layer of dimensions 10x1. Just like in the previous network, this involves taking the dot product and then feeding the output through a softmax activation function. The output is subtracted by the target, and we can begin backpropagation.

### 4.3.2 Backpropagation

The process of the first layer of backpropagation is identical to that of the previous network. Once the weights are obtained from this operation, the weights are fed backwards through the pooling layer and multiplied by the derivative of the ReLU function (which happens to be the step function, so only positive derivatives are maintained, while negative derivatives become very close to zero). Backpooling is as simple as restoring the dimensions of the feature maps before they were pooled, pre-allocating with zeros, and feeding each output value back into the index that was passed through the pooling layer on the forward pass. This is illustrated in Figure 12.

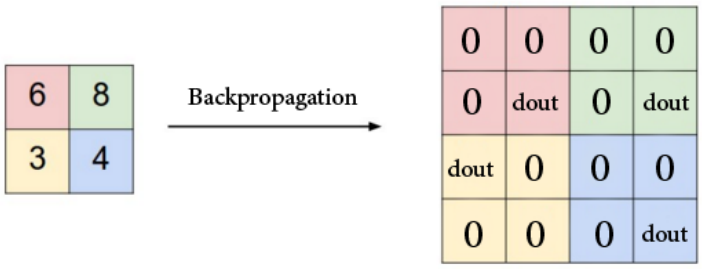



Figure 12: Backpooling allows only the derivatives of the values that were pooled to pass. All others go to zero. Source.

Backpropagation for convolution is a two-step process that involves first calculating the change in error with respect to the output, which gets fed further up the network for backpropagation, and the change in error with respect to the filters, which make up the gradient components. Both of these steps involve convolution. To calculate the outputs, I rotated each of the 8x5x5 filters 180 degrees and performed 2D convolution with the stack of 16 14x14 outputs from the backpooling layer (which is mostly filled with zeros). To calculate the gradient, I treated the 16 14x14 feature maps as a single input of 16 channels, and treated the 8 1x14x14 original outputs as the filters, and performed 3D convolution[12]. By padding the inputs with 2 layers of zeros, I obtained 16 5x5 gradient components.

Backpooling again resulted in dimensions of 8 28x28 feature maps primarily made up of zeros, ReLU reduced any negative gradients to near-zero, and backpropagation of the final convolution layer returned 8 5x5 gradient components. Then, all that was left was to multiply each component by the learning rate (selected to be .0001 through testing) and subtract from the current weights.

### 4.3.3 Training and Testing

The process for training and testing a convolutional neural network is the same as before. I implemented stochastic gradient descent, and began by training it on a single test example. After this proved successful, I moved on to training it on a small data set. I will discuss more about the difficulties I encountered with this network in Section 5.

# 5 Results and Discussion

With regards to the simpler, feed-forward network, I had quite a bit of success. Using randomly initialized weights, 7 epochs with a batch size of 500 and a learning rate of .001, I obtained a maximum 95% training accuracy and 89% test accuracy, while the minimum losses were .16 for training and .61 for test. The training and test graphs from this network are shown in Figures 13 and 14.

[12] 3D convolution is the same process as 2D convolution only you must convolve over every channel in the input image. The formula for this can be found here.

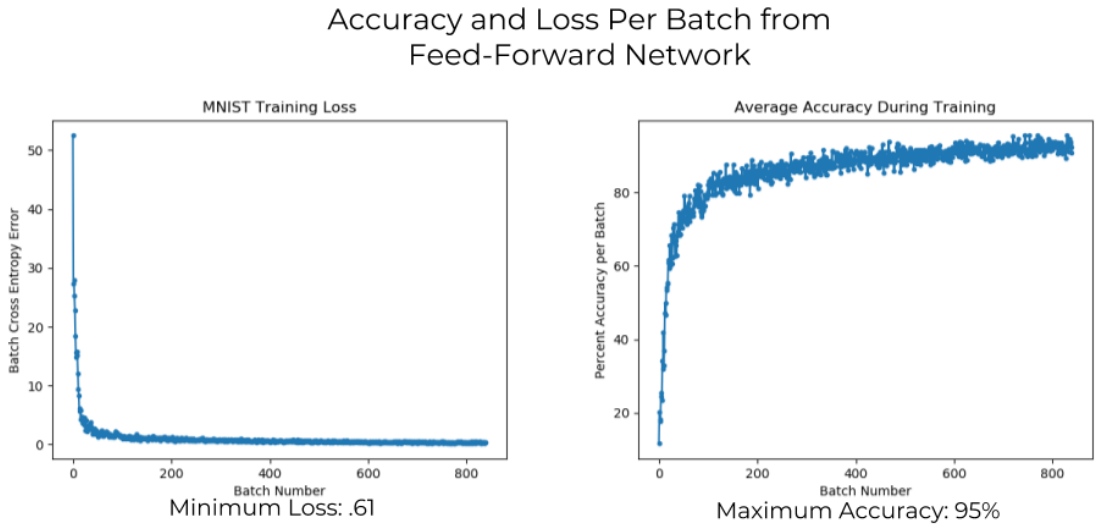



Figure 13: Results of training the feed-forward network on the full training set. The graph on the left shows the loss quickly converging around zero, while the accuracy rose fairly quickly to 80%, and continued to make slight improvements until capping at about 95%. Each point corresponds to the averaged loss and accuracy of one batch of 500 images.

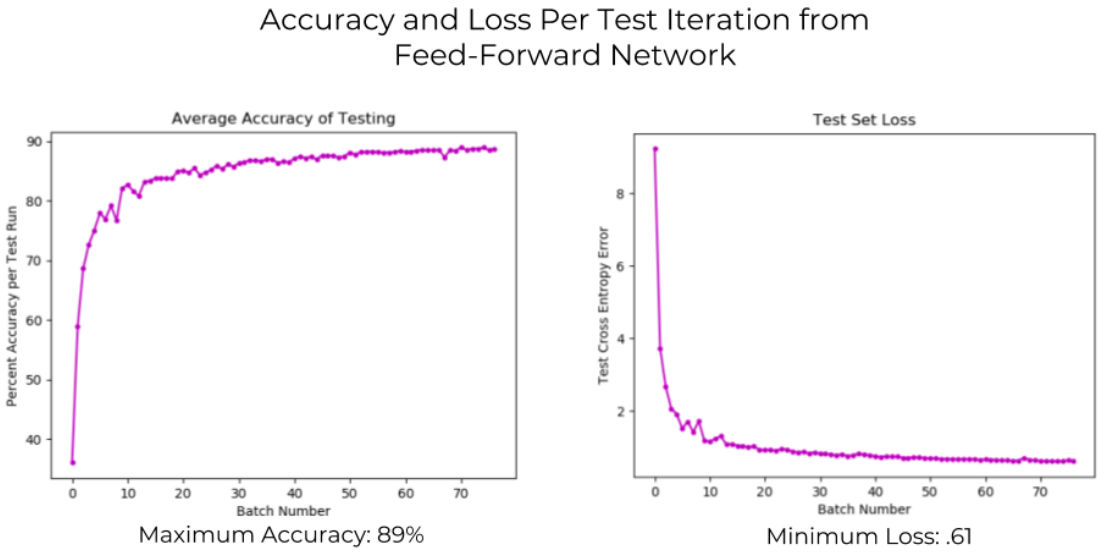



Figure 14: Results of testing the feed-forward network on the full test set. Each point corresponds to the average loss and accuracy over the entire test set of 1000 images at one iteration. Note that this graph is on the same x-axis scale as the training graphs in Figure 13, so even after the training loss converged, test accuracy and loss continued to increase and decrease respectively.

These graphs demonstrate that the network was well-tuned, because there is not a lot of noise, and even after randomly initializing the weights, the loss and accuracy values converge pretty quickly. Also, the fact that the test set accuracy is about 90% accurate demonstrates that the network is doing well at generalizing the information it trained on, not specializing to only recognize the training data set.

Another method of validation involved recording the indices of each image that the network guessed correctly and visualizing them. A small selection of these images are shown in Figure 15. Overall, this provided a great benchmark for my convolutional neural network.

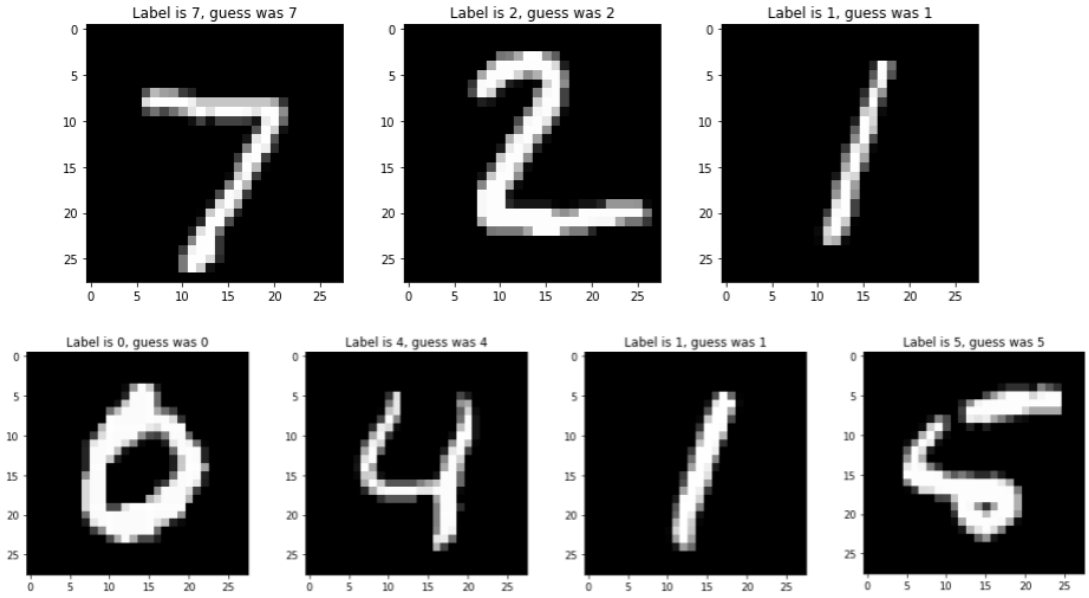



Figure 15: A selection of MNIST numbers that were properly classified by the neural network.

With regards to the convolutional neural network implementation, I encountered numerous technical difficulties. The derivations and process for implementation of a convolutional neural network are documented, but not clearly, and so after spending a very long time reading every resource I came across, I finally scraped out an implementation. However, I ran in to some issues with overflow caused by numbers railing to zero or becoming exceptionally close to zero, and if I had more time I would continue to debug this network. At the moment it appears to be pretty unstable, as it will converge near zero loss for hundreds of iterations, but at some point the loss suddenly spikes and the gradient generates a lot of `NaN`s. I tried to combat this by replacing my softmax activation function with a softmax LSE function, and with doing a similar thing to the cross-entropy error function (which has a $\log(x)$ in it), however that only delayed the instability, and eventually the runtime error would reappear on a simple multiplication. I would really like to continue to tune this network, since I believe it is close to working. This was evidenced by the network's ability to achieve 100% accuracy in reproducing one through five distinct test images within 20-30 iterations. So far, I have managed to obtain up to 80% test accuracy without the network fully converging on the full MNIST data set.

Figure 16 shows an example of the instability of the network. In this example, I was testing the network's ability to reproduce 5 different training images (just like testing the network on a single point but slightly more robust). The loss has actually very nearly reached zero, but at around batch number 67 the loss spikes rapidly and the network breaks.

Figure 17 shows one more example, where the network was able to complete its training of recognizing two points after passing each of the two possible numbers through the network 50 times. Notice that there is some overfitting present in this particular run of the network, where the network learned very quickly, then became "too confident" in its choices, and became less accurate for a time as a result.

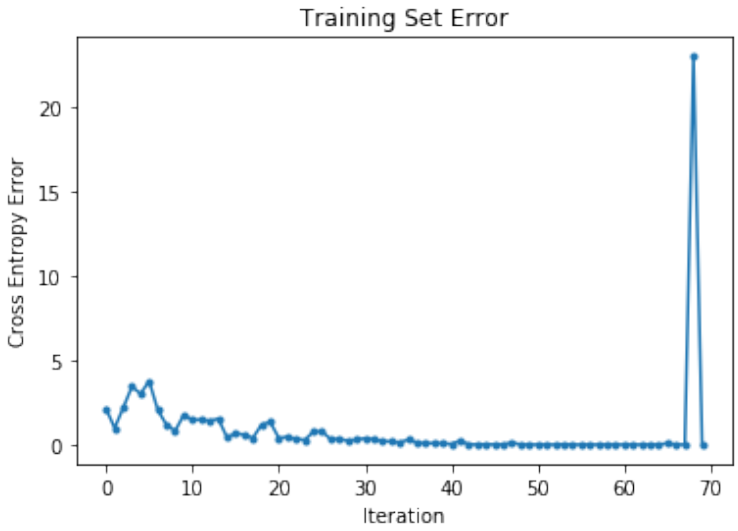



Figure 16: A plot of the the loss of the convolutional neural network over 70 iterations. The weights were randomly initialized and the error quickly converged to very close to zero for 30 to 40 iterations before suddenly spiking and generating `NaN`s. The last point dropping to zero is a result of the network breaking and the implementation of my code. In reality the value is `NaN`.

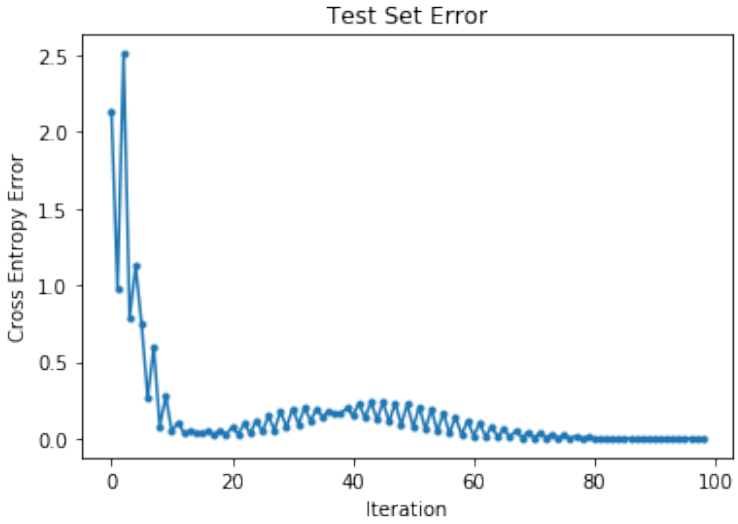



Figure 17: A plot of the loss function of the convolutional neural network when being trained on 2 input images. The slight oscillation corresponds to the network always viewing one number than the other. Also, the middle section between 20 and 60 iterations is a result of the network overfitting the data, where it becomes overly confident in its results and actually becomes less accurate.

# 6 Conclusion

Overall, this project had a lot of successes. The feed-forward network could be randomly initialized and train within 20 minutes on my laptop, and the convolutional network also performed calculations very quickly, and could be trained to accurately classify small groups of numbers, despite its instability. I was successful in writing all of my code from scratch, using only the Numpy library and default Python methods to accomplish my goal. This had the added benefit of reinforcing my programming skills, though the end result could have

been a bit more polished with some foresight (For instance, I did not make use of a class-based program structure). I also learned about the power of testing seemingly simple cases early on, to ensure that there are no errors in the code or gaps in conceptual understanding. I would have done this even more, perhaps starting with an even more simplified convolutional network to verify that each component was functioning as I expected it to, before building back up to the full MNIST implementation.

At the beginning, I was aware that neural networks are a complicated and somewhat ambiguous topic, but upon reflection I definitely did not realize the absolute magnitude of things I would be learning about (I simply could not include all of the information within this paper). The feed-forward neural network turned out to be an ambitious endeavour on its own, and understanding and executing an implementation involving convolutions also involved learning about a lot of new concepts.

# 7 Source Code

Click here to go to my Github repo for this project.

The relevant code is concentrated in `feedforward.py`, `stochastic.py`, and `CNN.ipynb`, though more visualizations can be found in the Jupyter notebooks.