UNCOVERING THE MATHEMATICS BEHIND NEURAL NETWORKS

DYLAN ZAPZALKA

Abstract. This paper explores the basics of neural networks as well as some fundamental machine learning topics from a more mathematical point of view. After first reviewing machine learning from an abstract point of view, the paper focuses on artificial neurons and artificial neural networks. Topics about neural networks include the backpropagation algorithm, as well as the Universal Approximation Theorem for single layered neural networks. We then demonstrate both the power and limitations of feed-forward neural networks through several examples showing their accuracy in image classification problems.

1. Introduction

Machine learning is the field within the study of computer science, statistics, and mathematics that gives computer algorithms the ability to learn with respect to an increased amount of data. Although mathematics creates a foundation that machine learning could not exist without, many people who build machine learning models with Python libraries such as TensorFlow and PyTorch don't actually know the mathematics behind the models they create. However, without the knowledge of its mathematical concepts, trying to create an efficient model consists of nothing more than monotonous hours of guesswork that have a high probability of leading nowhere. Mathematical fluency is not essential for building a machine learning model, but it is essential for building an efficient machine learning model.

Throughout this paper, we will explore the mathematics behind neural networks. However, before moving forward with the mathematics specifically pertaining to neural networks, we will do a quick recap of the basics of machine learning. The following is mainly based on the textbook "An Introduction to Statistical Learning" [1] and the book "Neural Networks and Deep Learning" [2].

2. Recap of Machine Learning Basics

2.1. **Prediction Function.** Throughout this section, we will use the following example to demonstrate the prediction function and other fundamental concepts. Suppose we want to predict the prices of houses in Fargo, ND, denoted as Y, using some m amount of predictors, denoted as $X_1, X_2, ..., X_m$. We now assume there exists some relationship between $X = (X_1, X_2, ..., X_m)$ and Y such that $Y = \varphi(X) + \epsilon$, where φ is some fixed function of X; ϵ is an irreducible error term that has a mean of zero; and ϵ is independent of X. Therefore, if we want to predict Y using X, we need to find the prediction function $\hat{\varphi}$ that represents an estimate of φ. The prediction function is what we will use to predict Y using X.

Our goal with machine learning is to get $\hat{Y} = \hat{\varphi}(X)$ as close to Y as possible. To do this, we will first have to find a way to train $\hat{\varphi}$ such that $\hat{Y} \approx Y$.

2.2. **Cost Function.** Before we can start training $\hat{\varphi}$, we need to make an assumption about the functional form of φ. For example, we may assume that φ is a linear function:

$$\varphi(X) = w_0 + w_1X_1 + w_2X_2 + ... + w_mX_m.$$

After we have made the assumption about the functional form of φ, we need to estimate the parameters, also known as the weights, such that

$$Y \approx w_0 + w_1X_1 + w_2X_2 + ... + w_mX_m.$$

To be able to estimate the weights of our assumed function, we need a way to measure how well some set of weights, $w_0, w_1, w_2, ..., w_m$, perform on a training set of data. This is exactly what the cost function does. For example, one of the most commonly used cost functions for regression problems is the mean squared error (MSE), where n denotes the number of training examples:

$$MSE = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \hat{\varphi}(x_i))^2.$$

Throughout this paper, we will denote the cost function as J. Now we need to find a way of minimizing our cost function so that we can find the optimal weights given our training data set. To do this, we will use a method called gradient descent.

2.3. **Gradient Descent and Learning Rate.** Gradient descent takes advantage of one of the fundamental properties of the gradient of a function: the gradient of some function $J(w)$, denoted $\triangledown J$ and defined as $\triangledown J(w) = (\frac{\partial J}{\partial w_0}, \frac{\partial J}{\partial w_1}, ..., \frac{\partial J}{\partial w_m})$, always points in the direction of the steepest ascent. Therefore, we should be able to get the direction of steepest descent by $-\triangledown J$.

Now that we have the gradient, we can perform gradient descent. Gradient descent is the process of updating the weights by subtracting it from the gradient multiplied by some learning rate, denoted as α, as shown below:

$$w := w - \alpha\triangledown J.$$

The learning rate is used to define how big of a step we want to take in the direction of the gradient during each update. Although simple, it has a large impact on how successfully our prediction function will train. If the α is too big, the weights will not converge to a local minimum. If α is too small, gradient descent will take an unreasonable amount of time to converge to the local minimum. It is the job of the person training the model to define α such that the weights converge at a quick rate relative to the given problem.

Therefore, as long as we assign α to an adequate value, we will eventually get our estimated weights for $\hat{\varphi}$ using gradient descent. Although the cost function measures how well a prediction function does with a training set, it is not an accurate indicator of how well our function will do in the real world. This is due to a process called overfitting.

2.4. **Overfitting, Underfitting, Bias, Variance, and Regularization.** Overfitting, also known as high variance, occurs when our prediction function pays too much attention to random noise within our training set. This can be due to not having enough training examples or having too many parameters in our prediction function, $\hat{\varphi}$. To fix this we can either add more training examples, decrease the amount of parameters in $\hat{\varphi}$, or we can implement regularization. Regularization is when we increase the cost function with respect to the value of the weights. This is typically done by adding the following to the end of the cost function:

$$J_{\text{regularized}} = J + \lambda \sum_j w_j^2,$$

where λ is called the regularization coefficient. If your model is doing very well on the training set but not well on the test set, your model is probably suffering from overfitting and high variance.

On the opposite side of the spectrum, we have underfitting, also known as high bias. Underfitting occurs when we our prediction function is unable to accurately assess any of the relationships between the predictors and the output. To fix this, we can either add more parameters or we can decrease the regularization coefficient. We can also add more training examples, but this is more than likely not going to help your model increase its accuracy significantly. You can tell if your model is suffering from underfitting and high bias if it is doing poorly both on the training data and the testing data.

2.5. **Normalization.**

The last basic subject we will talk about is normalization. We use normalization to speed up the process of gradient descent by getting the values of each parameter in our training set to approximately the same range and distribution. To do this, we subtract each value by the mean and divide it by the standard deviation:

$$\forall x_i \in X_i, x_i := \frac{x_i - \bar{x_i}}{\sigma_{x_i}}.$$

2.6. **Example.**

Now let's go back to the example of predicting the housing prices of houses in Fargo. Suppose that we want to predict the house prices, Y, given their square footage, denoted as X_1. In this example, we have a training set of size 28 [3] as is shown below in Figure 1.

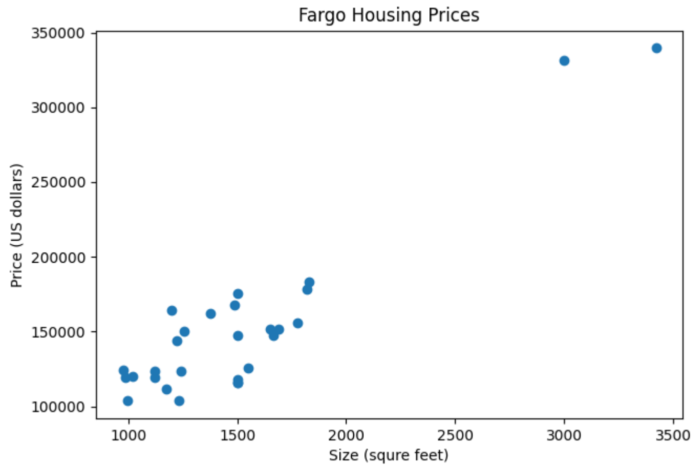

FIGURE 1. Fargo housing data set

We will assume that the functional form of φ is linear. Therefore, since we only have one parameter, the square footage, our prediction functions form will be $\hat{\varphi}(X_1) = w_0 + w_1 X_1$.

Now we must train the parameters w_0 and w_1 using the training data. Although the initial weights of the function should be randomized, for this example we will just set them each to the arbitrary number of 60. After tinkering with the model a bit, I decided to implement normalization as it decreased the amount of epochs (training cycles) needed to converge to the global minimum by an order of magnitude. With a learning rate of 0.1 and 100 iterations, we found it converged to the global minimum, as shown in Figure 2.

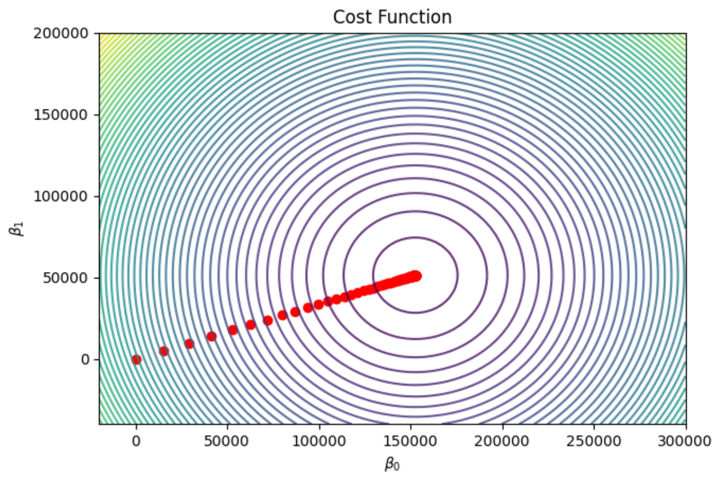

FIGURE 2. Parameters converging to the global minimum

Now that we have our parameters, we can complete our prediction function for the prices of houses in Fargo, given the square footage. Our function is shown in Figure 3 below.

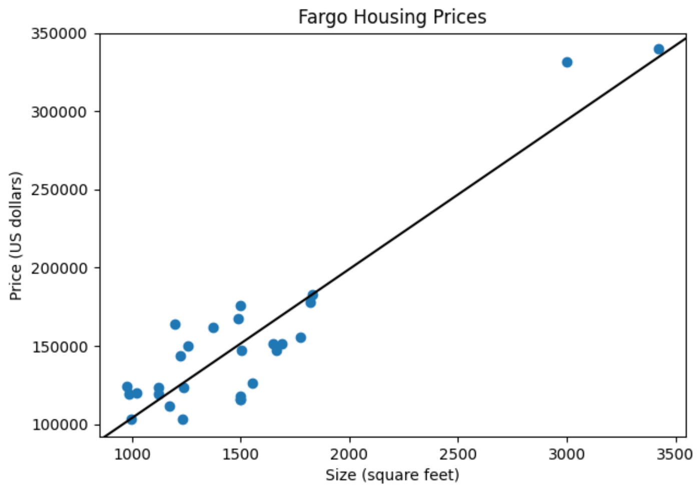

FIGURE 3. Final prediction function

Although this was one of the more basic examples of machine learning, all other supervised models follow a similar process, including classification models. However, one big difference you may find in more sophisticated models is the use of logistic regression. But instead of going through all of the other more advanced machine learning models, I believe it is best that we go straight to one of the most complex and widely used machine learning models today: neural networks.

3. Neural Networks

Although linear regression and other simple models work great for a wide variety of applications, there are many problems such as imagine recognition and weather prediction where these simpler models are not able to adequately capture any complex relationships between the predictors and the output. This is where neural networks come into play.

Before defining what a neural network is, it would be best to define the building blocks that make them up. Although there is a rich history of topics and ideas that helped form what we now consider the modern neural network, such as Aristotle's concept of Associationism that was developed at around 300BC [4], we will start much closer to the present time with the idea of the perceptron, created by Frank Rosenblatt in 1958 [5].

3.1. **Perceptrons.**

A perceptron is a function for binary classification problems that takes in a tuple of binary values, sums up each binary value by its corresponding weight, and outputs a 1 or 0 depending on if the sum is greater than some threshold. A more concrete definition is given below.

Definition. Let φ be some perceptron, $x \in \{0,1\}^n$, $w \in \mathbb{R}^n$, and $b \in \mathbb{R}$. Then φ is defined as follows:

$$\varphi(x) = \begin{cases} 0 \text{ if } \sum_{i=1}^{n} x_i w_i + b \leq 0 \\ 1 \text{ if } \sum_{i=1}^{n} x_i w_i + b > 0. \end{cases}$$

You can think of the perceptron as a decider that weighs a list of evidence to make a yes or no decision. For example, say I wanted to make a perceptron to decide if I am going to drive to a location or not. We can have three predictors represented by 1's for yes and 0's for no for the following questions: is it more than two miles, is there no traffic, and is there good parking? Of course distance is the main decider for whether or not I will drive, so the weight associated with that will be higher. However, traffic and good parking spots are typically less of a factor, so the weights for those two will be smaller.

Perceptrons can also be used to make elementary logic functions as well. For example, we will make the OR, NOR, AND, and NAND functions by changing the weights of the perceptron φ.

- To create the OR function, we can set $w_1 = 10$, $w_2 = 10$, and $b = -5$.
- To create the NOR function, we can set $w_1 = -10$, $w_2 = -10$, and $b = 5$.
- To create the AND function, we can set $w_1 = 10$, $w_2 = 10$, and $b = -15$.
- To create the NAND function, we can set $w_1 = -10$, $w_2 = -10$, and $b = 15$.

Although the perceptron may work well for the applications described above, it is very hard to train due to its binary nature. Preferably, we would like a small change in the predictors to result in a small change in the prediction. However, with the perceptron, a small change either does nothing to the output, or results in the output being flipped from a 0 to a 1 and vice versa. This is why we typically use more sophisticated models, such as the sigmoid neuron.

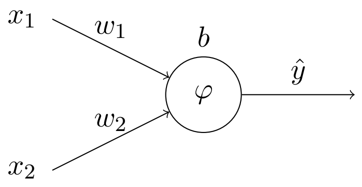

FIGURE 4. A basic perceptron model for creating logical functions

3.2. **Neurons.** The sigmoid neuron works much like the perceptron with a few small changes. Instead of inputting and outputting some binary values, the sigmoid neuron can input and output any values between 0 and 1.

Definition. Let φ be some sigmoid neuron, $x \in (0,1)^n$, $w \in \mathbb{R}^n$, and $b \in \mathbb{R}$. Then φ is defined as follows:

$$\varphi(X) = \sigma(\sum_{i=1}^{n} x_i w_i - b) \text{ where } \sigma = \frac{1}{1+e^{-z}}.$$

If you were to graph the sigmoid function (also known as the logistic function), σ, you will notice that it is a smooth curve. This continuity is the property we were looking for; that is, a small change in the input results in a small change in the output.

Although σ is a popular activation function, it does not need to be defined as the sigmoid function. As long as the function is non-linear and is computationally efficient, it can be used as an activation function. Some more activation functions are presented below [6].

Definition. The Tanh function is defined as

$$\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

Definition. The ReLU function is defined as

$$\sigma(x) = \begin{cases} x \text{ if } x \geq 0 \\ 0 \text{ if } x < 0. \end{cases}$$

Definition. The Exponential Linear Units function is defined as

$$\sigma(x) = \begin{cases} x \text{ if } x > 0 \\ \alpha e^x - 1 \text{ if } if \leq 0 \end{cases}$$

where α is some real number hyper-parameter usually set to 1.

Now that we have all of the necessary prerequisite knowledge, we can finally move onto the main topic of this paper.

3.3. **Neural Networks.** A neural network can be thought of as a network of neurons that form a complex, non-linear prediction function made from the linear combination and composition of activation functions. A neural network can be split into three parts: the input layer, the hidden layers, and the output layer. The input layer is comprised of input neurons that represent each predictor we want to include in our model. For example, supposed we have a 10×10 grey-scaled image. Then we would have 100 input neurons with each input neuron representing the gray-scaled value of some pixel. Next we have the hidden layers. The hidden layers are the layers comprised of neurons with non-linear activation functions in between the input and output layers. You can think of the hidden layers as the foundation of our neural network. Lastly, the output layer is simply the last layer of our network comprised of output neurons that give us the final output of the network. Figure 5 shows the neural network we will soon create to classify images.

Now that we know how to create a prediction function with a neural network, how do we train its parameters?

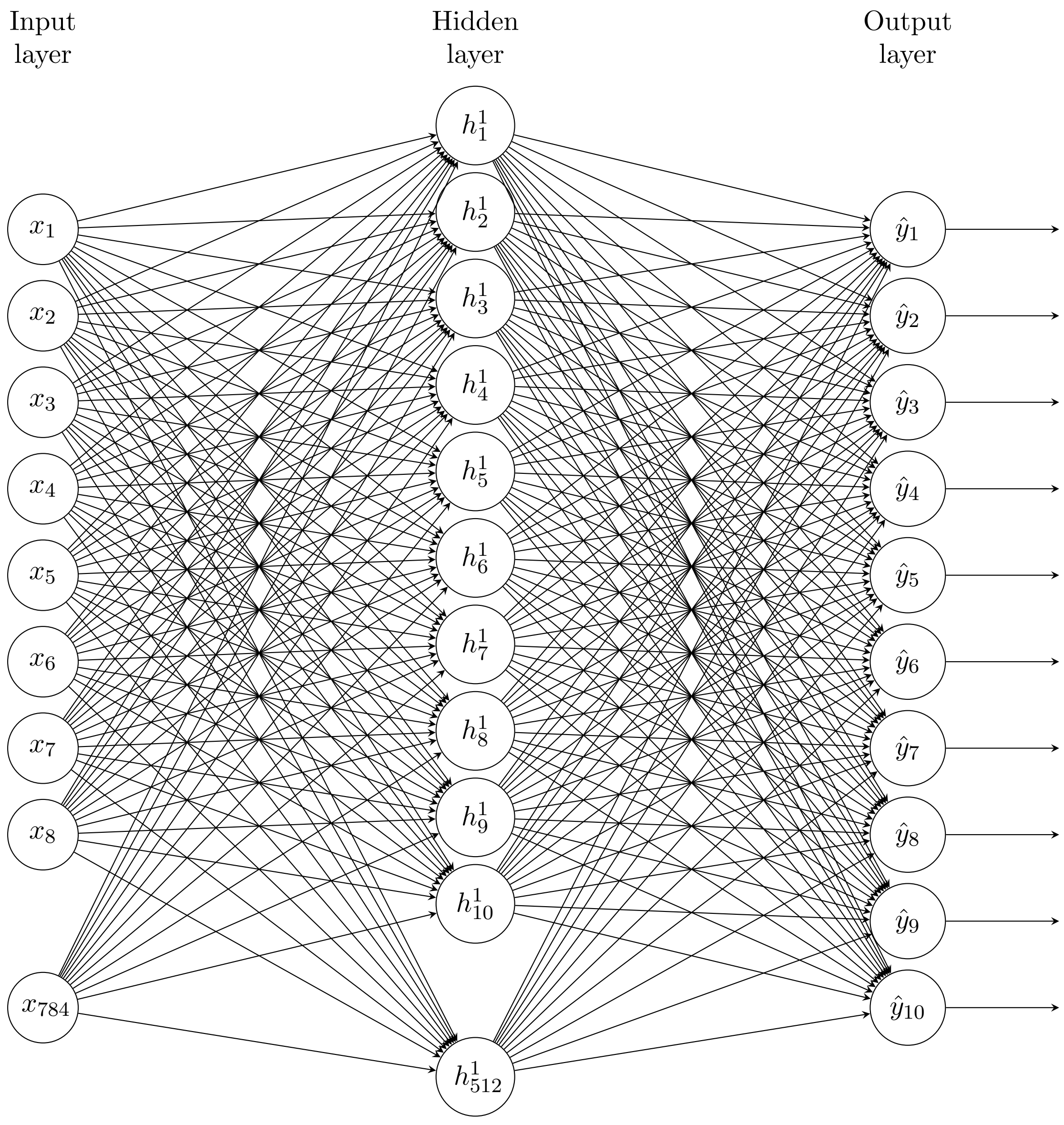

FIGURE 5. Neural network model for a 32×32 pixel digit classifier

3.4. **Backpropagation.** If you recall from Section 2.3, we used the gradient and the gradient descent algorithm to update the parameters so that they converge to a local minimum of the cost function which in turn increases the accuracy of our linear prediction function. The good news is, that is exactly what we do for non-linear neural networks. The bad news is, we still need some method to get the gradient of our neural network's cost function. However, there exists an algorithm called backpropagation which will allow us to calculate the gradient.

Before we can perform backpropagation, we need to make two assumptions about the cost function. The first assumption is that the cost function can be written as an average of cost functions for individual training examples, i.e.

$$J = \frac{1}{n}\sum_x J_x.$$

This is needed so we can compute $\frac{\partial C_x}{\partial w}$ and $\frac{\partial C_x}{\partial b}$ for a single training example so we don't need to compute $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$, which is very computationally expensive. Next, we must also assume that the cost function is a function of the outputs of our neural network, i.e.

$$C = C(\hat{y}).$$

Now that we have those assumptions out of the way, we must define some variables. First, we denote w_{jk}^l to be the weight for the connection from neuron k in layer $l-1$ to neuron j in layer l. Next, we denote b_j^l to be the bias for neuron j in layer l and we denote a_j^l to be the activation for neuron j in layer l. Now let us denote an intermediate quantity, z_j^l, to be the value inside of the activation function for neuron j in layer l. In other words, $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$. Lastly, we define $\delta_j^l = \frac{\partial C}{\partial z_j^l}$.

Now that we have have all of our needed assumptions and definitions, it is finally time to show the process of back propagation. We will start by giving four necessary equations for the backpropagation algorithm.

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \tag{1}$$

Proof. To get Equation 1, we will apply the chain rule to $\delta_j^L = \frac{\partial C}{\partial z_j^L}$:

$$\delta_j^L = \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L).$$

□

$$\delta_j^l = \sigma'(z_j^l)\sum_k w_{kj}^{l+1}\delta_k^{l+1} \tag{2}$$

Proof. To get Equation 2, we first need to apply the chain rule to $\delta_j^l = \frac{\partial C}{\partial z_j^l}$:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l}\delta_k^{l+1}.$$

Next, we need to substitute the partial derivative $\frac{\partial z_k^{l+1}}{\partial z_j^l}$.

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^l = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^l \Rightarrow$$

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l) \Rightarrow$$

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l) = \sigma'(z_j^l) \sum_k w_{kj}^{l+1} \delta_k^{l+1}.$$

□

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{3}$$

Proof. To get Equation 3, all we need to do is apply the chain rule to $\frac{\partial C}{\partial b_j^l}$.

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} = \delta_j^l.$$

□

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{4}$$

Proof. All we need to get Equation 4 is to apply the chain rule to $\frac{\partial C}{\partial w_{jk}^l}$.

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} a_k^{l-1} = a_k^{l-1} \delta_j^l.$$

□

With these four equations, we are able to perform the backpropagation algorithm to retrieve the gradient of the cost function.

For describing the backpropagation algorithm, I will use the matrix and vector versions of the four equations above. Not only does it greatly simplify the equations, but it also better represents how you would actually implement the algorithm inside a programming language such as Python or R.

Definition. Let X denote the matrix of the training example predictors where each row represents a training example and let X_j denote training example j. Next, let a^0 denote the input neurons, h^l denote the hidden neurons in hidden layer l, and let a^l and z^l denote the activation and it's intermediate values in the layer l such that all of these listed variables are represented as column vectors. Lastly, when performing a function on a vector or matrix such as $\sigma(X)$, we define it as performing $\sigma(X_{ij})$ for all i and j.

The following steps form what is known as the backpropagation algorithm:

(1) **Input:** The first and most simple step is to set the first layer of neurons to some training examples predictors. In other words, let $a^0 = X_j$ for the j'th training example.
(2) **Feedforward:** Compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$ for $l = 1, 2, 3, ..., L$.
(3) **Output Error:** Calculate the output error vector using Equation 1:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \Rightarrow \delta^L = \nabla_a C \odot \sigma'(z^L).$$

(4) **Backpropagate the Error:** Calculate δ^l for $l = 1, 2, 3, ..., L-1$ using Equation 2:

$$\delta_j^l = \sigma'(z_j^l)\sum_k w_{kj}^{l+1}\delta_k^{l+1} \Rightarrow \delta^l = \sigma'(z^l) \odot w^{l+1}\delta^{l+1}.$$

(5) **Gradient Calculation:** Now calculate the gradient of the cost function by using Equation 3 and Equation 4:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \Rightarrow \frac{\partial C}{\partial b^l} = \delta^l.$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \Rightarrow \frac{\partial C}{\partial w^l} = a^{l-1}(\delta^l)^T.$$

And that's all there is to the backpropagation algorithm. After running the algorithm, we will have obtained the gradient of the cost function, which we can now use to train our model using gradient descent:

Definition. For $l = 1, 2, ..., L$

$$\begin{cases} w^l := w^l - \alpha(a^{l-1}(\delta^l)^T) \\ b^l := b^l - \alpha(\delta^l). \end{cases}$$

Using everything learned above, we now have developed all of the necessary knowledge to create and train a neural network that can adequately perform some complex task, such as image recognition. Before moving onto examples to demonstrate the power of the mathematical concepts displayed above, we will cover an interesting theorem demonstrating the power of neural networks. This theorem, known as the Universal Approximation Theorem, shows that a single layered neural network can predict any continuous function within any degree of error greater than 0.

4. Universal Approximation Theorem

The Universal Approximation Theorem for neural networks states that for any continuous function, there exists some single layered neural network that can approximate it to any degree of error. The following section is based off of Cybenko's famous 1989 paper titled "Approximation by Superpositions of a Sigmoidal Function" [7].

For the proof, we are going to prove that single-layered neural networks using a sigmoidal activation function are universal approximators. A sigmoidal function is any function that approaches 1 as it goes to infinity and approaches 0 as it goes to negative infinity.

Definition. A function σ is defined as sigmoidal if

$$\sigma(x) = \begin{cases} 1 \text{ as } x \to +\infty \\ 0 \text{ as } x \to -\infty \end{cases} .$$

Recall from Section 3.2, the sigmoid activation function would be defined as sigmoidal. Now we will give a definition for discriminatory functions.

Definition. Let $\mu \in M([0,1]^n)$ be some signed regular Borel measure on $[0,1]^n$. We define σ to be discriminatory if

$$\int_{[0,1]^n} \sigma(w^T x + b)\, d\mu(x) = 0$$

for all $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$ implies that $\mu = 0$.

Now that we have the required definitions, we can move on to proving the required lemmas for our proof. First, we will show that single-layered neural networks using a discriminatory activation function are universal approximators.

Lemma 1. *Let $\varphi \in C([0,1]^n)$ and σ be some continuous discriminatory activation function. Also let $\epsilon \in \mathbb{R}$ such that $\epsilon > 0$. Then there exists some single layered neural network, defined as $\hat{\varphi} = \sum_{i=1}^{n} \alpha_i \sigma(w_i^T x + b_i)$, such that $|\hat{\varphi}(x) - \varphi(x)| < \epsilon$ for all $x \in [0,1]^n$.*

Proof. Let $S \subset C([0,1]^n)$ where S is the set of functions of the form of $\sum_{i=1}^{n} \alpha_i \sigma(w_i^T x + b_i)$ and σ is some continuous discriminatory function. We wish to show that the closure of S is $C([0,1]^n)$. To do this, we will use a proof by contradiction.

Let us assume that the closure of S is not $C([0,1]^n)$. Then the closure of S is $R \subset C([0,1]^n)$. Trivially, $C([0,1]^n)$ is a normed vector space and S is a subspace of $C([0,1]^n)$. As a consequence of the Hahn-Banach Theorem, we know that there exists some some bounded linear functional, $L \in C([0,1]^n)$, such that $L(R) = L(S)$ and $L \neq 0$.

Then by the Riesz Representation Theorem, L can be written in the form

$$L(h) = \int_{[0,1]^n} h(x)\, d\mu(x)$$

where $\mu \in M([0,1]^n)$ for all $h \in C([0,1]^n)$. Since $\sigma(w^T x + b) \in R$ for any w and b, and $R \subset C([0,1]^n)$, then

$$\int_{[0,1]^n} \sigma(w^T x + b)\, d\mu(x) = 0.$$

Since σ is a discriminatory function, $\mu = 0$ which implies that $d\mu = 0$. Therefore $L = 0$, which is a contradiction since we assumed that $L \neq 0$. Thus, the closure of S must be $C([0,1]^n)$.

□

Lemma 2. *Any sigmoidal function is is discriminatory.*

Proof. To show that any sigmoidal function is discriminatory, we must show that

$$\int_{[0,1]^n} \sigma(w^T x + b)\, d\mu(x) = 0$$

for some sigmoidal function σ implies that $\mu = 0$. Therefore, let's assume that $\int_{[0,1]^n} \sigma(w^T x + b)\, d\mu(x) = 0$.

By the definition of the sigmoidal function we know that

$$\sigma(\lambda(w^T x + b) + \rho) = \begin{cases} \to 1 \text{ for } w^T x + b > 0 \text{ as } \lambda \to +\infty \\ \to 0 \text{ for } w^T x + b < 0 \text{ as } \lambda \to +\infty \\ = \sigma(\rho) \text{ for } w^T x + b = 0 \text{ for all } \lambda. \end{cases}$$

Therefore, as $\lambda \to +\infty$, the functions of the form $\sigma(\lambda(w^T x + b) + \rho)$, denoted as σ_λ, converge to the following function

$$\omega(x) = \begin{cases} = 1 \text{ for } w^T x + b > 0 \\ = 0 \text{ for } w^T x + b < 0 \\ = \sigma(\rho) \text{ for } w^T x + b = 0. \end{cases}$$

Now let us define some hyperplane $\Pi_{w,b} = \{x \mid w^T x + b = 0\}$, the open half-space $H_{w,b} = \{x \mid w^T x + b > 0\}$, and the other half-space $-H_{w,b} = \{x \mid w^T x + b < 0\}$.

By Lebesgue's Dominated Convergence Theorem, given a sequence of measurable functions (f_n) that converges pointwise everywhere as $n \to \infty$ and $|f_n| \leq g$ for some integrable g and all n, then f is integrable and the following statement holds:

$$\int_S f \, d\mu = \lim_{n\to\infty} \int_S f_n \, d\mu.$$

Therefore, for any w, b, ρ, the following statement holds:

$$0 = \int_{[0,1]^n} \sigma(\lambda(w^T x + b) + \rho) \, d\mu(x) = \int_{[0,1]^n} \lim_{\lambda\to\infty} \sigma_\lambda(x) \, d\mu(x) = \int_{[0,1]^n} \omega(x) \, d\mu(x) =$$
$$\int_{\Pi_{w,b}} \sigma(\rho) \, d\mu(x) + \int_{H_{w,b}} 1 \, d\mu(x) + \int_{-H_{w,b}} 0 \, d\mu(x) = \sigma(\rho)\mu(\Pi_{w,b}) + \mu(H_{w,b}) \Rightarrow$$
$$\mu(\Pi_{w,b}) = \mu(H_{w,b}) = 0.$$

We will now show that the measure of all half-planes being equal to zero implies that the measure itself is equal to zero. To do this, let's fix w and define some linear functional F by some bounded measurable function h such that

$$F(h) = \int_{[0,1]^n} h(w^T x) \, d\mu(x).$$

We then define h to be an indicator function on the interval $[b, \infty)$, where $h(z) = 1$ if $z \geq b$ and $h(z) = 0$ if $z < b$. Using the definition of the indicator function of h,

$$F(h) = \int_{[0,1]^n} h(w^T x) \, d\mu(x) = \mu(\Pi_{w,-b}) + \mu(H_{w,-b}) = 0.$$

If we were to define h to be an indicator function on the interval (b, ∞), where $h(z) = 1$ if $z > b$ and $h(z) = 0$ if $z \leq b$, we would get a similar result:

$$F(h) = \int_{[0,1]^n} h(w^T x) \, d\mu(x) = \mu(H_{w,-b}) = 0.$$

By the property of linearity, $F(h) = 0$ for any indicator function of any interval. Thus, for any simple function, $F(h) = 0$. And since simple functions are dense in $L^\infty(\mathbb{R})$, $F = 0$.

Thus, given

$$F(\sin + i\cos) = \int_{[0,1]^n} \cos(m^T x) + i \sin(m^T x) \, d\mu(x) = \int_{[0,1]^n} e^{im^T x} \, d\mu(x) = 0$$

for all m.

Hence, $\mu = 0$. Thus, σ is discriminatory. □

Now using Lemma 1 and Lemma 2, we can prove Theorem 4.1.

Theorem 4.1. *Let $\varphi \in C([0,1]^n)$ and σ be some continuous sigmoidal activation function. Also let $\epsilon \in \mathbb{R}$ such that $\epsilon > 0$. Then there exists some single layered neural network, defined as $\hat{\varphi} = \sum_{i=1}^{n} \alpha_i \sigma(w_i^T x + b_i)$, such that $|\hat{\varphi}(x) - \varphi(x)| < \epsilon$ for all $x \in [0,1]^n$.*

Proof. Using Lemma 1, we know that the statement holds true for when σ is discriminatory. Also, from Lemma 2, we know that all sigmoidal functions are discriminatory. Therefore, Theorem 4.1 must be true. □

We have now shown that single layered neural networks can approximate any continuous function to any degree of error greater than zero, given that we use a sigmoidal activation function such as the sigmoid function. This proof leaves out other very popular activation functions that are not sigmoidal such as the ReLU function. However, we can still indirectly use this proof to prove that other activation functions work as well by proving that they are discriminatory.

Lastly, the Universal Approximation Theorem may raise the question: if we can predict any function with a singled layer neural network, why is there a need for multi-layered neural networks? The simple answer to that is although a single-layered neural network can develop any function a multi-layered neural network can, the single layered neural networks usually take far more neurons and training time to do the same job that a multi-layered neural network can do.

Now it's time to demonstrate the power of neural networks by developing different neural networks using the techniques described throughout all of the previous sections.

5. Neural Network Examples

5.1. **Example 1.** For the first example, we will be training the neural network in Figure 5 to recognize handwritten digits using the MNIST dataset. For our models, we will be using the ReLU activation function and a learning rate of 0.001. The accuracy for both the training and the testing will taken after 32 epochs, which has been enough for the values to converge on the cost function.

As we can see from Figure 6, even a single layered neural network can do a very accurate job of predicting basic images, such as single digit numbers. The best performing model, the one with 1024 neurons, was able to get an accuracy of 98.34%.

If we wanted to improve the model even further, we could add multiple more hidden layers or add something called convolutional layers. However, I believe it would be best to demonstrate how these additions benefit image classification models by using a more complex dataset.

5.2. **Example 2.** For this example, we will be training a neural network to classify images of 10 different objects in colored 32×32 pictures using the CIFAR10 dataset. Since it is colored and has 32×32 pixels, the input layer will consist of $32 \times 32 \times 3$ neurons, where each neuron represents the red, green, or blue value of one of the pixels. And since we

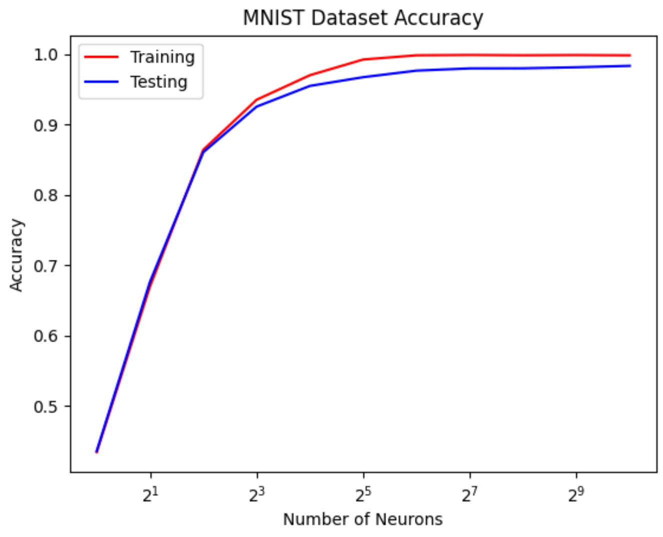

FIGURE 6. MNIST accuracy for single layered networks

are classifying 10 different types of images, we will have 10 output neurons to represent each different type of image. Figure 7 shows the results of single layered neural networks of different sizes and activation functions.

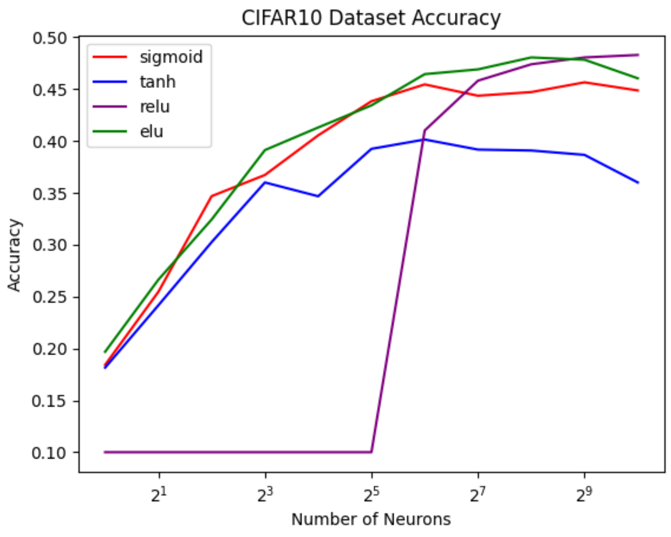

FIGURE 7. CIFAR10 accuracy for single-layered networks

For this example, the model did much worse due to the added complexity of the problem. The best model was the one with the ReLU activation functions and 1024 neurons, which was able to classify the images with an accuracy of roughly 48%. Even with three hidden layers, a standard feed-forward neural network was only able to get up to a little over 50% accuracy as shown in Figure 8.

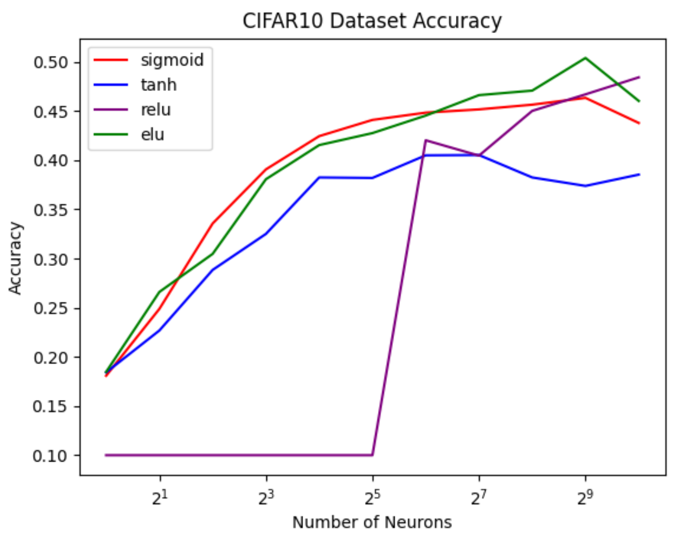

FIGURE 8. CIFAR10 accuracy for multi- layered networks

Therefore, it is clear that in order to get an adequate accuracy for more complex problems, other advanced neural networks are needed.

Using more advanced techniques, I was able to get the accuracy shown below.

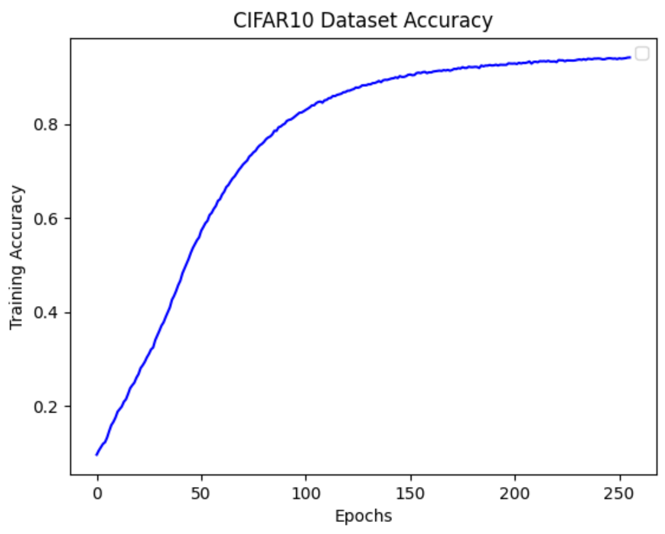

FIGURE 9. CIFAR10 training accuracy for an advanced network

For the testing accuracy, I was able to get 88.63% using techniques such as regularization, dropout, convolutional layers, and max pooling. Thus, it is important to keep in mind that although neural networks are powerful tools that should be in any data scientist's arsenal, much of their power and usability comes from additional features such as the ones described above.

References

[1] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: with Applications in R.* Springer, 2021.

[2] M. A. Nielsen, "Neural networks and deep learning," 2019.

[3] S. Narula and J. Wellington, "Linear regression and the minimum sum of relative errors," 1977.

[4] H. Wang and B. Raj, "On the origin of deep learning," 2017.

[5] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," pp. 386–408, 1958.

[6] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," 2018.

[7] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

Advisor: Indranil SenGupta

dylan.zapzalka@ndsu.edu

Department of Mathematics, North Dakota State University, PO Box 6050, Fargo, ND 58108-6050, USA