An Overview Of Artificial Neural Networks for Mathematicians

Leonardo Ferreira Guilhoto

Abstract

This expository paper first defines what an Artificial Neural Network is and describes some of the key ideas behind them such as weights, biases, activation functions (mainly sigmoids and the ReLU function), backpropagation, etc. We then focus on interesting properties of the expressive power of feedforward neural networks, presenting several theorems relating to the types of functions that can be approximated by specific types of networks. Finally, in order to help build intuition, a case study of effectiveness in the MNIST database of handwritten digits is carried out, examining how parameters such as learning rate, width, and depth of a network affects its accuracy. This work focuses mainly on theoretical aspects of feedforward neural networks rather than providing a step-by-step guide for programmers.

Contents

1 Introduction

Artificial Intelligence has been a popular topic in science fiction and news articles for at least a few decades, and now many algorithms of this category are indeed incorporated in the everyday life of people with self-driving cars, automatically generated image captions on search engines, recommendation algorithms, and even job hirings. Despite the popularity of this term, it does not have a precise definition other than "a program that does something smart" with the meaning of "smart" changing throughout the years. More recently, this term has overwhelmingly been used to refer to *machine learning* algorithms, which can also be called *data driven* algorithms, since they are nothing more than algorithms which adjust some of their parameters in response to a data set. Out of these algorithms, one type is currently especially popular and is the topic of this expository paper: *Artificial Neural Networks*.

Algorithms that fall under this term are more often than not used to mimic some sort of action which could be executed by a human being, but for which we lack the mathematical tools to explicitly model in a fully applicable way. A classical example of this is image classification: when you see a picture of a dog, for instance, you brain immediately recognizes the animal and would most likely rarely mistake it for, say, a lizard. Although biology does indeed study in depth how this recognition process works, we are yet to be able to explicitly write down a static program which is capable of analyzing the picture of an animal and correctly classifying what species it belongs to. Such a program would have to take into consideration the particular configuration of each individual pixel in relation to all others, and be invariant through translation, rotation, etc. Take for example a simple 1,280 by 960 pixel image: such a file would contain information about 1,228,800 pixels, each of which is composed of 3 components (red, blue, and green) which are numbers within some specified range (for the sake of simplicity, assume it is normalized to be some number between 0 and 1), resulting in 3,686,400 total parameters. Therefore, a program that attempts to classify this image would in practice be calculating some function $f : [0,1]^{3,686,400} \to \{a | a \text{ is an animal species}\}$. Given this huge dimensional dependence, and arguably arbitrary nature of features that define each possible animal, writing from scratch a program that accomplishes this feat would be virtually impossible.

This is where artificial neural networks (henceforth also referred to as just "neural networks") come in. Instead of trying to figure out explicitly what this function is, we attempt to approximate it with another function $F : [0,1]^{3,686,400} \to \{a | a \text{ is a species from some pre-determined collection of animals}\}$, in which we do not initially attempt to write down a close approximation, but rather allow the function to "learn" from labeled data (meaning several pictures of animals with their a priori classified species). You can think of this process as starting out with a function F_0 and then completing several training stages (often called *training epochs*), with a new function F_i being obtained after every i^{th} stage. The result of this is a sequence of functions $\{F_i\}$ which we hope has the property

$$\lim_{n\to\infty} F_n = f.$$

In short, the network starts out computing a function which might be a terrible approximation of f, and compares the classification done by this function with images previously classified by human beings. By programing the network to adapt in order to obtain progressively better classification rates, we end up with a program that correctly labels new images in a satisfying and consistent manner. In a similar spirit, Section 4 of this paper uses a neural network to associate pictures of digits into their appropriate categories.

This method has achieved such successful results over the past decade that neural networks have gained considerable strength with programmers and played an ever-increasing part in society. Nowadays, *deep learning* is being used to determine life-changing actions such as medical diagnoses, natural resource management, mortgage rates and more, but our theoretical knowledge of why and how they work so well in many cases is still in its infancy. As the influence of neural networks grows, it is important to advance our understanding of them in order to counter undesirable outcomes which have been noted in the past and persist today such as networks incorporating racism and other forms of social bias from their training data. To quote the computer scientist Ali Rahimi: "*I'm okay using technology I don't understand. I got here on an airplane and I don't fully understand how airplanes work... but I take comfort knowing that there is an entire field of aeronautics dedicated to creating that understanding.*" With that in mind, mathematicians do have a great role to play in order to advance this theory in a formalized and rigorous way.

This paper is aimed towards mathematicians and students of mathematics who have at least some background in multivariable calculus. Knowledge of some basic functional analysis is helpful for understanding the proof of the Universal Approximation Theorem, but this background is not assumed and all the required results are stated beforehand.

Although there are many types of artificial neural networks such as *Recurrent* (often used for speech and audio recognition) and *Convolutional* (often used for image classification), this paper focuses generally on

feedforward networks, which will be defined in the following section. This is perhaps the simplest form of a neural network, which offers a great introduction to the field while still retaining an incredibly rich range of applications and intellectual questionings.

The structure of this paper is as follows: first, we introduce the concept of feedforward neural networks and many ideas tied to this structure such as activation functions, backpropagation and more; second, we explore the expressive power of neural networks, meaning that we study what types of functions can be approximated by different types of networks and then pursue bounds on the size of these networks; finally, a case study of neural networks applied to the MNIST database of handwritten digits is included, in order to help the reader gain intuition behind some practical aspects of neural networks. In a broad sense, the first part presents its contents mostly in a conceptual manner (although with appropriate notation and sketches of some proofs); the second one engages in formal mathematics, with rigorous definitions and proofs; and the third mostly presents interesting examples and phenomena that one might encounter when initially implementing neural networks.

2 An Overview of Feedforward Neural Networks

2.1 Structure

2.1.1 Nodes And Layers

A feedforward neural network is an algorithm which works in waves, or layers. In each layer, there are a given number of nodes, which are sometimes called perceptrons, or even neurons (by analogy with the biological brain). Nodes are simply representations of numbers stored in the program. Starting with the input layer, all the way to the output layer, the values stored (called the *activation* of the node), say, in the n^{th} layer will be used to compute the values stored in the nodes in the $(n+1)^{th}$ layer. Mathematically, the activations stored in a layer with k nodes can be represented by a vector of dimension k.

In order to refer to the structure of a specific neural network, the term *depth* is commonly used to refer to the number of hidden layers in the network. Hence, when media sources (and often specialist) use the term "deep network" or "deep learning" they simply mean that a specific network has a large number of layers. In analogy to this term, another one, less commonly used, can also be found in literature: *width*. Width usually refers to how many nodes there are in each hidden layer of the network. This term is well defined for networks in which all hidden layers have the same number of nodes (the width is then this number), but not so much in networks in which the number of nodes varies from layer to layer. A visual representation of a feedforward network is presented in Figure 1.

This system is analogous to biological networks of neurons, in which one neuron firing up (transmitting an electric signal) will fire up (propagate the electric signal) nearby neurons. A significant difference between these two systems is that in biological networks, a neuron is either activated or not, while in artificial neural networks it may have a continuum of activations, such as a real number between 0 and 1 (like in sigmoid networks), a non-negative number (like in ReLU networks), or even an imaginary number! In the beginnings of artificial neural networks, some algorithms attempted to implement a binary activation system, only allowing the values on nodes to be either 0 or 1. However, this approach significantly hindered the efficiency of the learning process.

2.1.2 Weights, Biases and Activation Functions

Given the values stored in the n^{th} layer of the network (with, say, k nodes), how are the values of the next layer (with, say, j nodes) calculated? Each neuron in the n^{th} layer is given a specific 'connection strength' to each node in the next layer. This 'connection strength' is mathematically represented by what is called a *weight*, which is a real number. For notation, set $w^n_{a,b} \in \mathbb{R}$ as the weight between the a^{th} node in the $(n-1)^{th}$ layer to the b^{th} node in the n^{th} layer. In order to represent all the weights connecting these two layers, we may use a matrix

$$W_n := \begin{bmatrix} w^n_{1,1} & \dots & w^n_{1,k} \\ \vdots & \ddots & \vdots \\ w^n_{j,1} & \dots & w^n_{j,k} \end{bmatrix}.$$

We may also wish to add what is called a "bias", or "threshold", which is a constant $b \in \mathbb{R}^j$, and an "activation function" which is a function $\sigma : \mathbb{R} \to \mathbb{R}$ that converts the value calculated by the weights and

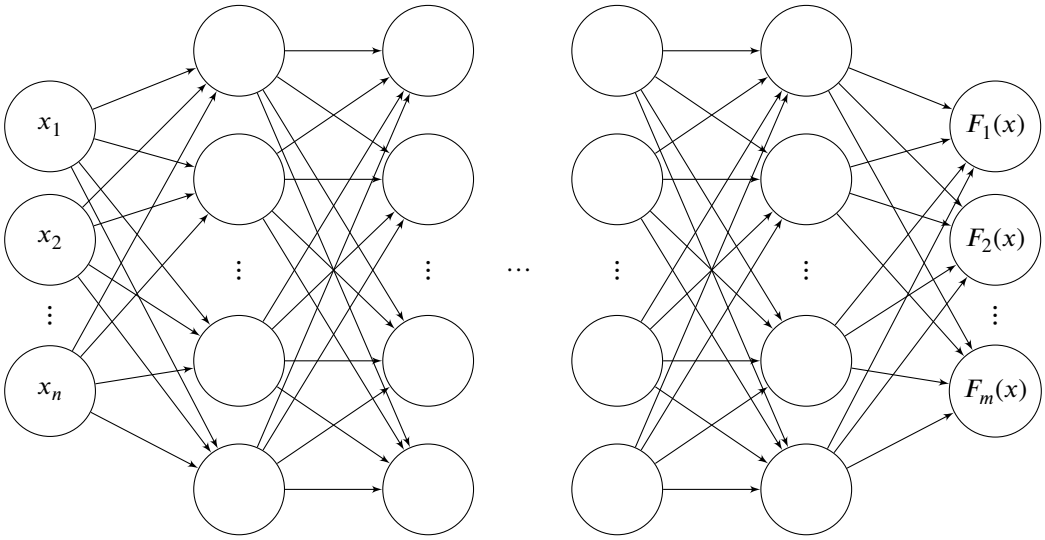

Figure 1: A visual representation of a feedforward network which approximates some function $f : \mathbb{R}^n \to \mathbb{R}^m$ by computing the function $F(x) = (F_1(x), \cdots, F_m(x))$. In this approach, the network is shown as a directed weighted graph. Note that here we adopt the notation $x = (x_1, \cdots, x_n)$.

biases into something which can be stored in the network. In the end, in order to calculate the state A_n of the n^{th} layer, we use the formula

$$A_n = \sigma(W_n A_{n-1} + b_n) = \sigma\left(\begin{bmatrix} w^n_{1,1} & \dots & w^n_{1,k} \\ \vdots & \ddots & \vdots \\ w^n_{j,1} & \dots & w^n_{j,k} \end{bmatrix}\begin{bmatrix} a^{n-1}_1 \\ \vdots \\ a^{n-1}_k \end{bmatrix} + \begin{bmatrix} b^n_1 \\ \vdots \\ b^n_j \end{bmatrix}\right).$$

In the above equation, we employ the following notation for matrices

$$\sigma\left(\begin{bmatrix} a_{1,1} & \dots & a_{1,k} \\ \vdots & \ddots & \vdots \\ a_{j,1} & \dots & a_{j,k} \end{bmatrix}\right) = \begin{bmatrix} \sigma(a_{1,1}) & \dots & \sigma(a_{1,k}) \\ \vdots & \ddots & \vdots \\ \sigma(a_{j,1}) & \dots & \sigma(a_{j,k}) \end{bmatrix}.$$

As the computations are carried out along the network's layers, the final function F calculated by a network of depth N is

$$F(x) = \sigma(W_N \sigma(\dots \sigma(W_2 \sigma(W_1 x + b_1) + b_2) \dots) + b_N).$$

2.2 Learning Process

Section 3 uses the structure introduced above to determine what types of functions can be approximated by neural networks. However, in a real world scenario, these networks are used to estimate functions which we do not know how to write down analytically (if we did, there would be no need to go though the trouble of setting up a neural network). Although this paper does not go in depth regarding the learning process, a rough outline of the most common method of training neural networks will be outlined here. It cannot be stressed enough that even though this paper mostly explores the properties of expressivity of neural networks, most emphasis and problems regarding the use of these algorithms rely on the training process, and how to obtain the best results given a finite training data set. A neural networks is, after all, a data driven algorithm.

2.2.1 Cost Function

Suppose we want to approximate a function $f : \mathbb{R}^n \to \mathbb{R}^m$, and our network currently computes a function $F : \mathbb{R}^n \to \mathbb{R}^m$. In order to determine how well a certain prediction given by the algorithm is, we may establish a cost function, which measures the discrepancy of two functions. To do that, we can feed some input x of which we know the value of $f(x)$ (this would be called a *label data point*) to the network and see what result it computes. In a broader sense, we may see how well the network performs on a testing set $(x_1, \dots, x_N)$ in which we know all the values for $f(x_1), \dots, f(x_N)$. There is more than one choice for

such a function, but consider the following one, which is often referred to as the "Mean Square Error" or "Quadratic Cost Function"

$$C := \frac{1}{2N}\sum_{i=1}^{N} ||f(x_i) - F(x_i)||^2, \tag{1}$$

where $||\cdot||$ represents the usual $\mathbb{R}^m$ norm

$$||y|| := \sqrt{y_1^2 + \cdots + y_m^2} \qquad \text{for } y = (y_1, \ldots, y_m) \in \mathbb{R}^m.$$

This function, then, becomes large when our network approximates f badly, and small when the approximation is accurate. Additionally notice that if we set $C_x := \frac{1}{2N}||f(x) - F(x)||^2$, we have that

$$C = \sum_{i=1}^{N} C_{x_i}.$$

This property will be important in the algorithm described in Section 2.2.3.

2.2.2 Gradient Descent

Gradient descent is a standard method used in optimization problems. It relies on concepts from multivariable calculus.

Definition 2.1. The gradient of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ at a point $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ is a vector in $\mathbb{R}^n$ of the form

$$\nabla f(x) := \left(\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_n}(x)\right). \tag{2}$$

It is a well known result that, given a point $x \in \mathbb{R}^n$, the gradient at that point indicates the direction of steepest ascent. Given that f is differentiable at x, it can be approximated linearly, and therefore the vector $-\nabla f(x)$ indicates the direction of steepest descent of the function f at the point x.

In order to obtain the minimum value of the function, the gradient descent strategy tells us to start at a given $x_0 \in \mathbb{R}^n$, calculate the value of $\nabla f(x_0)$, and then proceed to calculate a new point $x_1 := x_0 - \lambda \nabla f(x_0)$, where $\lambda > 0$ is called the *learning rate*. We then repeat this process, creating a sequence $\{x_i\}$ defined by our initial choice of x_0, the learning rate λ, and the rule: $x_{i+1} = x_i - \lambda \nabla f(x_i)$ for any $i \in \mathbb{N}$. This sequence continues until we approach a region close to our desired minimum.

The method of gradient descent when taken continuously over infinitesimally small increments (that is, taking the limit $\lambda \to 0$) and in regards to a "nice" function usually converges to a local minimum. However, depending on the location of the initial x_0, the local minimum achieved may not be the global minimum of the function. Furthermore, since when carrying out calculations on an unknown function we must take discrete steps (which vary in length depending on the learning rate), we are not even guaranteed a local minimum but rather may oscillate close to one, or even 'jump' past it altogether if the learning rate is too big. Still, even with these possible complications, gradient descent is a surprisingly successful method for many real life applications and is the most standard method of training for feedforward neural networks and many other machine learning algorithms.

Given that our cost function indicates how poorly our neural network approximates a given function, by calculating the gradient of the cost function with respect to the weights and biases of the network and adjusting these parameters in the direction opposite to the gradient, we will decrease our error and therefore lead us closer to an adequate network (in most cases).

2.2.3 Backpropagation

The explanation in this section is a simple introduction to the method of backpropagation in feedforward neural networks. The proof of some equations will only be sketched, as they are not related to the mathematics of the expressive power of networks, which forms the bulk of this paper. For a more comprehensive understanding of the backpropagation algorithm, the second chapter of the online book *Neural Network and Deep Learning* (available at: http://neuralnetworksanddeeplearning.com/) is a great reference. Much of the ideas developed in this section and in Section 4 are based on this book. It should also be noted that the notation in this section seems rather complicated, but the math behind the process basically relies only on repeated applications of the chain rule, and is not too complicated if the reader is familiar with this

rule from multivariable calculus. Having said that, the notation *is* cumbersome, and much attention should be paid to the innumerous indices in the formulas.

The big challenge of applying gradient descent to neural networks is calculating the partial derivatives of the cost function with respect to each individual weight and bias (namely, for every $\frac{\partial C}{\partial w^l_{i,j}}$ and $\frac{\partial C}{\partial b^l_j}$). This is where backpropagation comes in. This algorithm first tells us how to calculate these values for the last layer of connections, and with these results then inductively goes "backwards" through the network, calculating the partial derivatives of each layer until it reaches the first layer of the network. Hence the name "backpropagation". For now, we think of the cost function as taken over a training set consisting of a single labeled data point x.

The property mentioned at the end of Section 2.2.1 now becomes extremely important, as the algorithm developed in this section only works if we know the values of all activations in the network for a given input. Therefore, in fact, we only know how to calculate ∇C_x for a given labeled data point x. However, because of the property, we use

$$\nabla C = \nabla(\sum_{i=1}^{N} C_{x_i}) = \sum_{i=1}^{N} \nabla C_{x_i}, \tag{3}$$

which means that we can execute this process for each data point and then add the values of the gradient together. This is important because employing backpropagation in a large training set for every training epoch becomes a very computationally expensive approach, so instead we select a few elements from the training set[1], calculate the gradient, update the network and repeat the process until the network arrives at satisfactory results.

In a previous section, we defined A_n to be the vector representing the activations of the nodes in the n^{th} layer of the network. However, for the purposes of this section, it will be useful for us to consider what the values sent by the previous layer were *before* the activation function was applied. Consider

$$z^l_j := \sum_k w^l_{j,k} a^{l-1}_k + b^l_j \qquad \text{so that} \qquad a^l_j = \sigma(z^l_j) \qquad \text{and} \qquad A^l = \sigma(Z^l).$$

In the previous equation, $Z^l := \sum_k z^l_k e_k$ is a vector with entries corresponding to the values z^l_j (e_i are the standard basis vectors). Additionally, consider the quantity

$$\delta^l_j := \frac{\partial C}{\partial z^l_j} \qquad \text{and} \qquad \Delta^l := \sum_k \delta^l_k e_k. \tag{4}$$

These values will be useful for propagating the algorithm backwards through the network and are directly related to $\frac{\partial C}{\partial w^l_{i,j}}$ and $\frac{\partial C}{\partial b^l_j}$ by the chain rule, since

$$\frac{\partial C}{\partial w^l_{i,j}} = \frac{\partial C}{\partial z^l_j}\frac{\partial z^l_j}{\partial w^l_{i,j}} = \delta^l_j a^{l-1}_i \qquad \text{and} \qquad \frac{\partial C}{\partial b^l_j} = \frac{\partial C}{\partial z^l_j}\frac{\partial z^l_j}{\partial b^l_j} = \delta^l_j. \tag{5}$$

Since a^{l-1}_j is readily available for any node of the network, if we are able to calculate the value of the δ^l_j's we will have been successful in obtaining our gradient! Our first step is calculating this value for the last layer of the network, that is, δ^L_j for a network with L layers. It is not hard to notice that, since $a^L_j = \sigma(z^L_j)$, again by the chain rule

$$\delta^L_j = \frac{\partial C}{\partial a^L_j}\frac{\partial a^L_j}{\partial z^l_j} = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j). \tag{6}$$

Using the Mean Square Error function, noticing that $F(x) = A^L = (a^L_1, \dots, a^L_k)$ and letting $f(x) = (y_1, \dots, y_k)$ we get that

$$\delta^L_j = (a^L_j - y_j)\sigma'(z^L_j), \tag{7}$$

which can easily be calculated by a computer if we know how to calculate σ' (which should be true for any practical activation function). In order to obtain this result in vector form, it is common to write

[1] A very common form of selecting these elements is randomly sampling from all available labeled data. A short note about this is included in Section 4, when *Stochastic Gradient Descent* is mentioned.

$$\Delta^L = \nabla_{A^L} \odot \sigma'(Z^L), \tag{8}$$

where $\nabla_{A^L} := (\frac{\partial C}{\partial a_1^L}, \dots, \frac{\partial C}{\partial a_k^L})$ is the gradient of C taken with respect to the elements of A^L and $\odot$ is the *Hadamard Product*, which multiplies two matrices (or vectors) elementwise, defined as

$$\begin{bmatrix} a_{1,1} & \dots & a_{1,k} \\ \vdots & \ddots & \vdots \\ a_{j,1} & \dots & a_{j,k} \end{bmatrix} \odot \begin{bmatrix} b_{1,1} & \dots & b_{1,k} \\ \vdots & \ddots & \vdots \\ b_{j,1} & \dots & b_{j,k} \end{bmatrix} := \begin{bmatrix} a_{1,1}b_{1,1} & \dots & a_{1,k}b_{1,k} \\ \vdots & \ddots & \vdots \\ a_{j,1}b_{j,1} & \dots & a_{j,k}b_{j,k} \end{bmatrix}.$$

Now we will only need to "propagate" this backwards in the network in order to obtain δ_j^{L-1}. In order to do so, apply the chain rule once again

$$\begin{aligned} \delta_j^{L-1} &= \frac{\partial C}{\partial z_j^{L-1}} \\ &= \nabla_{Z^L} C \cdot \frac{\partial Z^L}{\partial z_j^{L-1}} \\ &= \sum_i^k \frac{\partial C}{\partial z_i^L} \frac{\partial z_i^L}{\partial z_j^{L-1}} \\ &= \sum_i^k \delta_i^L \frac{\partial z_i^L}{\partial z_j^{L-1}}. \end{aligned}$$

If we focus on the term $\frac{\partial z_i^L}{\partial z_j^{L-1}}$, we find that

$$\begin{aligned} \frac{\partial z_i^L}{\partial z_j^{L-1}} &= \frac{\partial(\sum_k w_{i,k}^L a_k^{L-1} + b_i^L)}{\partial z_j^{L-1}} \\ &= \frac{\partial(\sum_k w_{i,k}^L \sigma(z_k^{L-1})^+ b_i^L)}{\partial z_j^{L-1}} \\ &= \frac{\partial(w_{i,j}^L \sigma(z_j^{L-1}))}{\partial z_j^{L-1}} \\ &= w_{i,j}^L \sigma'(z_j^{L-1}), \end{aligned}$$

which, again, can be easily calculated by a computer given the network. Therefore

$$\delta_j^{L-1} = \sum_i^k \delta_i^L w_{i,j}^L \sigma'(z_j^{L-1}). \tag{9}$$

This formula tells us how to calculate any δ_j^l in the network, assuming we know Δ^{l+1}. Since (8) tells us how to jump start Δ^L in the last layer of the network, we are done, and the algorithm is successful.

Summarizing what was done in this section, we first defined $\delta_i^l := \frac{\partial C}{\partial z_j^l}$, which has direct relationships with our desired values of $\frac{\partial C}{\partial w_{i,j}^l}$ and $\frac{\partial C}{\partial b_j^l}$. We then wrote down what the δ_i^L's should be for the last layer of the network, and finally developed a way to calculate all the δ_i^l's, given that we know what the values of δ_i^{l+1} are. Thus, by propagating this method backwards through the layers of the network we are able to find all our desired partial derivatives, and can therefore calculate the value of ∇C as a function of the weights and biases of the network and execute the method of gradient descent.

3 The Expressive Power of Feedforward Neural Networks

3.1 Universal Approximation

3.1.1 Useful Definitions and Theorems from Functional Analysis

Many of the proofs related to universal approximation rely on results from functional analysis. Most of these theorems have very standard proofs, which can be found in almost any textbook on functional analysis. For the sake of conciseness, the theorems used later in the paper will be stated, but their proofs will be omitted. The reader can find proofs for the following results in sources such as [Bre10].

Theorem 3.1. *(**Lebesgue Dominated Convergence Theorem**) Let X be a measure space, μ be a Borel measure on X, $g : X \to \mathbb{R}$ be L^1 and $\{f_n\}$ be a sequence of measurable functions from $X \to \mathbb{R}$ such that $|f_n(x)| \leq g(x)$ for all $x \in X$ and $\{f_n\}$ converges pointwise to a function f. Then f is integrable and*

$$\lim_{n\to\infty} \int f_n(x)d\mu(x) = \int f(x)d\mu(x). \tag{10}$$

Theorem 3.2. *(**Hahn-Banach Theorem - Geometric Form**) Let V be a normed vector space and $A, B \subset V$ be two non-empty, closed, disjoint and convex subsets such that one of them is compact. Then there exists a continuous linear functional $f \not\equiv 0$, some $\alpha \in \mathbb{R}$ and an $\epsilon > 0$ such that $f(x) \leq \alpha - \epsilon$ for any $x \in A$ and $f(y) \geq \alpha + \epsilon$ for any $y \in B$.*

We focus on a corollary derived from this theorem.

Corollary 3.3. *Let V be a normed vector space over $\mathbb{R}$ and $U \subset V$ be a linear subspace such that $\overline{U} \neq V$. Then there exists a continuous linear map $f : V \to \mathbb{R}$ with $f(x) = 0$ for any $x \in U$, and $f \not\equiv 0$.*

Proof. Let $z \in V \backslash \overline{U}$. By the previous theorem (and noting that the set $\{z\}$ is compact), there exists $f : V \to \mathbb{R}$ and $\alpha > 0$ such that $f(x) < \alpha$ for any $x \in \overline{U}$ and $f(z) > \alpha$, which means that

$$f(x) < \alpha < f(z) \qquad \text{for all } x \in \overline{U}.$$

Since f is linear, and $\overline{U}$ is a subspace, this means that, for any $x_0 \in \overline{U}, \lambda \in \mathbb{R}$ we have that

$$f(\lambda x_0) = \lambda f(x_0) < \alpha,$$

which means that

$$f(x_0) < \frac{\alpha}{\lambda} \quad \text{for all } \lambda > 0 \qquad \text{and} \qquad f(x_0) > \frac{\alpha}{\lambda} \quad \text{for all } \lambda < 0,$$

which implies that $f(x) = 0$ for all $x \in \overline{U}$ and $f(z) > \alpha > 0$. □

Definition 3.4. Given a topological space Ω, we define $C(\Omega) := \{f : \Omega \to \mathbb{R} | f \text{ is continuous}\}$.

Theorem 3.5. *(**Riesz Representation Theorem**) Let Ω be a subset of $\mathbb{R}^n$ and $F : C(\Omega) \to \mathbb{R}$ be a linear functional on the space of continuous real functions with domain on Ω. Then there exists a signed Borel measure μ on Ω such that for any $f \in C(\Omega)$, we have that*

$$F(f) = \int_\Omega f(x)d\mu(x). \tag{11}$$

We now move onto some definitions which are specific to the problem of universal approximation in neural networks.

Definition 3.6. For $f : \mathbb{R} \to \mathbb{R}$ an activation function, set

$$\Sigma_n(f) = span\{f(y \cdot x + \theta) | y \in \mathbb{R}^n, \theta \in \mathbb{R}\}. \tag{12}$$

In the above equation, $y \cdot x$ represents the standard dot product in $\mathbb{R}^n$.

The set $\Sigma_n(f)$ consists of all the functions that can be calculated by a neural network with a single hidden layer and activation function f.

Definition 3.7. Let Ω be a topological space and $f : \mathbb{R} \to \mathbb{R}$. We say that a neural network with activation function f is a universal approximator on Ω if $\Sigma_n(f)$ is dense in $C(\Omega)$, the set of continuous functions from Ω to $\mathbb{R}$.

One could, in theory, modify the above definition to be such that the set of functions generated by a neural network with activation function f is dense in $C(\Omega)$. This would relate more closely to what we mean by universal approximation and would enclose the previous definition (which assumes the network has only one hidden layer). However, for historical reasons, this is the definition which prevailed, and in no way does it hinder any of our results so it shall be the one employed.

Definition 3.8. Let n be a natural number. We say an activation function $f : \mathbb{R} \to \mathbb{R}$ is n-discriminatory if the only signed Borel measure μ such that

$$\int f(y \cdot x + \theta) d\mu(x) = 0 \quad \text{for all } y \in \mathbb{R}^n, \text{ and } \theta \in \mathbb{R} \tag{13}$$

is the zero measure.

Definition 3.9. We say an activation function $f : \mathbb{R} \to \mathbb{R}$ is discriminatory if it is n-discriminatory for any n.

Definition 3.10. A function $f : \mathbb{R} \to \mathbb{R}$ is called a sigmoid if it satisfies the following two properties:

$$\lim_{x \to \infty} f(x) = 1 \quad \text{and} \quad \lim_{x \to -\infty} f(x) = 0.$$

Definition 3.11. Let n be a natural number. Then we define

$$I_n := [0,1]^n = \{x = (x_1, \cdots, x_n) \in \mathbb{R}^n | x_i \in [0,1] \text{ for any } i = 1, \cdots, n\}.$$

Definition 3.12. The Rectified Linear Unit (also denoted ReLU) is a function $\mathbb{R} \to \mathbb{R}$ defined by:

$$ReLU(x) := \max(0, x).$$

3.1.2 Statement and Proof of Universal Approximation Theorem for Sigmoid and ReLU Activation Functions

A paper published by George Cybenko in 1989, titled "*Approximation by Superpositions of a Sigmoidal Function*" sparked a series of works for the next few years which attempted to determine which activation functions lead to the Universal Approximation Property. In [Cyb89], he proves this for continuous sigmoid functions by presenting what are Theorem 3.13 and Lemma 3.14 in the present paper.

Among the papers published on Universal Approximation, perhaps the one containing the strongest result is [LLPS93], published in 1993. In it, it is proven that any activation function will lead to a network with universal approximation capacity **if and only if** this function is not a polynomial almost everywhere (in the sense of measure theory). Despite this being an elegant paper, we will focus on the work done by Cybenko, which has certain historical importance and contains relatively short proofs.

Theorem 3.13. *Let f be a continuous discriminatory function. Then a neural network with f as the activation function is a universal approximator.*

Proof. For the sake of contradiction, assume $\Sigma_n(f)$ is not dense in $C(I_n)$. It follows that $\overline{\Sigma_n(f)} \neq C(I_n)$. We then apply the corollary of the Hahn-Banach theorem to conclude that there exists some continuous linear functional $F : C(I_n) \to \mathbb{R}$ such that $F \neq 0$ but $F(g) = 0$ for any $g \in \overline{\Sigma_n(f)}$. By the Riesz Representation Theorem, there exists some Borel measure μ such that

$$F(g) = \int_{I_n} g(x) d\mu(x) \qquad \text{for all } g \in C(I_n).$$

However, since for any y and θ the function $f(y \cdot x + \theta)$ is an element of $\overline{\Sigma_n(f)}$, this means that for all $y \in \mathbb{R}^n$, and $\theta \in \mathbb{R}$ we have $\int f(y \cdot x + \theta) d\mu(x) = 0$, which means that $\mu = 0$ (since f is discriminatory) and therefore $F(g) = 0$ for any $g \in C(I_n)$. This contradicts the corollary of the Hahn-Banach theorem, and thus finishes the proof. □

Lemma 3.14. *All bounded, Borel measurable sigmoid functions are discriminatory.*

Proof. Let f be a bounded, Borel measurable sigmoid function, and assume that for a given measure $\mu \in M(I_n)$ we have that

$$\int_{I_n} f(y \cdot x + \theta) d\mu(x) = 0 \text{ for all } y \in \mathbb{R}^n, \text{ for all } \theta \in \mathbb{R}.$$

Our aim is to show that $\mu = 0$. In order to do that, consider the function γ, obtained by fixing $\lambda, \phi \in \mathbb{R}$ and taking the limit

$$\gamma(x) = \lim_{\lambda \to \infty} f(\lambda(y \cdot x + \theta) + \phi) = \begin{cases} 1 & \text{if } y \cdot x + \theta > 0 \\ f(\phi) & \text{if } y \cdot x + \theta = 0 \\ 0 & \text{if } y \cdot x + \theta < 0 \end{cases}.$$

By the Lebesgue Dominated Convergence Theorem, we have that

$$\int_{I_n} \gamma(x) d\mu(x) = \lim_{\lambda \to \infty} \int_{I_n} f(\lambda(y \cdot x + \theta) + \phi) \mu(x) = 0.$$

By using the information above, we derive that

$$\begin{aligned} \int_{I_n} \gamma(x) d\mu(x) &= \int_{H^+_{y,\theta}} 1 d\mu(x) + \int_{\Pi_{y,\theta}} f(\phi) d\mu(x) + \int_{H^-_{y,\theta}} 0 d\mu(x) \\ &= \mu(H^+_{y,\theta}) + f(\phi)\mu(\Pi_{y,\theta}) \\ &= 0, \end{aligned}$$

where

$$\begin{aligned} H^+_{y,\theta} &:= \{x \in I_n | y \cdot x + \theta > 0\}, \\ \Pi_{y,\theta} &:= \{x \in I_n | y \cdot x + \theta = 0\}, \\ H^-_{y,\theta} &:= \{x \in I_n | y \cdot x + \theta < 0\}. \end{aligned}$$

This is true for any choice of y, θ. Since this is true for any ϕ, and $f(\phi) \to 1$ as $\phi \to \infty$, we get that

$$\mu(H^+_{y,\theta}) + \mu(\Pi_{y,\theta}) = 0.$$

In a similar fashion, if we let $\phi \to -\infty$, we get that $f(\phi) \to 0$ and

$$\mu(H^+_{y,\theta}) = 0.$$

Now consider the functional $F : L^\infty(\mathbb{R}) \to \mathbb{R}$ defined as:

$$F(h) := \int_{I_n} h(y \cdot x) d\mu(x).$$

Since $L^\infty(\mathbb{R})$ is the space of bounded functions, this integral (and therefore the functional) is well defined for any $h \in L^\infty(\mathbb{R})$. If we take $\mathbf{1}_{[\theta,\infty)}$ as the indicator function of the interval $[\theta, \infty)$, we get that

$$F(\mathbf{1}_{[\theta,\infty)}) = \int_{I_n} \mathbf{1}_{[\theta,\infty)}(y \cdot x) d\mu(x) = \mu(H^+_{y,\theta}) + \mu(\Pi_{y,\theta}) = 0.$$

Similarly, for any open interval (θ, ∞), we have

$$F(\mathbf{1}_{(\theta,\infty)}) = \int_{I_n} \mathbf{1}_{(\theta,\infty)}(y \cdot x) d\mu(x) = \mu(H^+_{y,\theta}) = 0.$$

Using linearity, we get that $F(h) = 0$ for the indicator function h of any interval. It follows that $F(h) = 0$ for any simple function. Using that simple functions are dense in $L^\infty(\mathbb{R})$, it follows that $F = 0$. Additionally, since sin and cos are elements of $L^\infty(\mathbb{R})$, we can use that

$$\begin{aligned} F(\cos) + iF(\sin) &= \int_{I_n} (\cos(y \cdot x) + i \sin(y \cdot x)) d\mu(x) \\ &= \int_{I_n} (\cos(y \cdot x) + i \sin(y \cdot x)) d\mu(x) \\ &= \int_{I_n} e^{iy \cdot x} d\mu(x) \\ &= 0. \end{aligned}$$

This is true for all $y \in \mathbb{R}^n$, which means that the Fourier transform of μ is 0. This is only possible if $\mu = 0$, and we are done with the proof.

□

We have now proven that the universal approximation property holds for networks with continuous sigmoid functions. However, many state-of-the-art algorithms presently use ReLU as their activation function. Although general results such as [LLPS93] apply to this function, we take a different approach, and, in the spirit of the work done by Cybenko, prove that the ReLU is 1-discriminatory, and then use Lemma 3.16 to show that this implies the universal approximation property for functions with inputs of finite dimension. To the best of the author's knowledge, this is the first time this approach is taken to prove that networks with ReLU activation function are universal approximators.

Lemma 3.15. *The ReLU function is* 1*-discriminatory.*

Proof. Let μ be a signed Borel measure, and assume the following holds for all $y \in \mathbb{R}$ and $\theta \in \mathbb{R}$:

$$\int ReLU(yx + \theta) d\mu(x) = 0.$$

We want to show that $\mu = 0$. For that, we will construct a sigmoid bounded, continuous (and therefore Borel measurable) function from subtracting two ReLU functions with different parameters. In particular, consider the function

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \in [0, 1] \\ 1 & \text{if } x > 1 \end{cases}. \tag{14}$$

Then any function of the form $g(x) = f(yx + \theta)$ with $y \neq 0$ can be described as

$$g(x) = ReLU(yx + \theta_1) - ReLU(yx + \theta_2) \tag{15}$$

by setting $\theta_1 = -\theta/y$ and $\theta_2 = (1 - \theta)/y$. If $y = 0$, then instead set

$$g(x) = f(\theta) = \begin{cases} ReLU(f(\theta)) & \text{if } f(\theta) \geq 0 \\ -ReLU(-f(\theta)) & \text{if } f(\theta) \leq 0 \end{cases}$$

Which means that for any $y \in \mathbb{R}, \theta \in \mathbb{R}$

$$\begin{aligned} \int f(yx + \theta) d\mu(x) &= \int (ReLU(yx + \theta_1) - ReLU(yx + \theta_2)) d\mu(x) \\ &= \int ReLU(yx + \theta_1) d\mu(x) - \int ReLU(yx + \theta_2) d\mu(x) \\ &= 0 - 0 = 0. \end{aligned}$$

By the previous lemma, f is discriminatory, and therefore, $\mu = 0$.

□

Lemma 3.16. *If $\Sigma_1(f)$ is dense in $C([0, 1])$ then $\Sigma_n(f)$ is dense in $C([0, 1]^n)$*

Proof. We use the fact that the span of the set $\{g(a \cdot x) \mid a \in \mathbb{R}^n, g \in C([0,1])\}$ is dense in $C([0,1]^n)$. That is, given any function $h \in C([0,1]^n)$ and $\epsilon > 0$ there exist functions g_k in $C([0,1])$ such that

$$|h(x) - \Sigma_{k=1}^{N} g_k(a_k \cdot x)| < \epsilon/2.$$

If we now examine each function $g_k(a_k \cdot x)$, and use the assumption that $\Sigma_1(f)$ is dense in $C([0,1])$, we conclude that for any such function, there exists a sum of functions such that

$$|g_k(a_k \cdot x) - \Sigma_{i=1}^{N_k} f(y_{k,i} \cdot x + \theta_{k,i})| < \epsilon/2k.$$

By applying the triangle inequality, we get that

$$\begin{aligned} |h(x) - \Sigma_{k=1}^{N}\Sigma_{i=1}^{N_k} f(y_{k,i} \cdot x + \theta_{k,i})| &< |h(x) - \Sigma_{k=1}^{N} g_k(a_k \cdot x)| + k(\epsilon/2k) \\ &< \epsilon/2 + \epsilon/2 \\ &= \epsilon. \end{aligned}$$

This shows we can get arbitrarily close to any function in $C([0,1]^n)$ by using functions in $\Sigma_n(f)$.

□

As a direct consequence of Theorem 3.13 and previous lemmas, we arrive at the desired result:

Corollary 3.17. *(**Universal Approximation Theorem**) Neural Networks with either ReLU or continuous sigmoid activation functions are universal approximators.*

3.2 Effective Versions of the Universal Approximation Theorem

We have now proved that a Neural Network with a single hidden layer and sufficiently many nodes is capable of approximating any continuous function. However, this design differs drastically from the usual architecture used in Neural Networks, in which a multitude of layers are employed (thus rending the fashionable term "deep learning"). Can we prove that a network with bounded width retains the Universal Approximation Property? The following simple result achieves this goal, at the cost of using a deeper structure (more layers) for the network.

Theorem 3.18. *Let $f : [0,1]^d \to [0,1]$ be the function calculated by a single layer ReLU neural network of input dimension d, output dimension 1 and n hidden nodes. Then there exists another ReLU neural network, with n hidden layers and width $d+2$, which calculates the same function f.*

The proof of this Theorem uses the strategy of using d of the nodes in each layer to store the initial input, one node to calculate each of the nodes in the original network, and the last node to store the value of the sum of the functions.

Proof. Since f is generated by a single layer ReLU network with n hidden nodes, it is of the form

$$f(x) = ReLU(b + \sum_{i=1}^{n} w_i ReLU(g_i(x))),$$

where each g_i is of the form $g_i(x) = y_i \cdot x + b_i$ with $y_i \in \mathbb{R}^d$ and $b_i \in \mathbb{R}$. Since $[0,1]^d$ is compact and any function generated by a ReLU network is continuous, the term $\sum_{j=1}^{i} w_j ReLU(g_j(x))$ achieves a minimum for any $i \leq n$ and therefore there exists a number $T > 0$ such that

$$T + \sum_{j=1}^{i} w_j ReLU(g_j(x)) > 0$$

for any $x \in [0,1]^d$ and any $i \leq n$.

Now consider a new network in which each hidden layer uses d nodes to copy the original inputs. Additionally each i^{th} hidden layer is also equipped with another node that computes the function $ReLU(g_i(x))$ from the input nodes copied to the previous layer, and a last node that computes

$$ReLU(T + \sum_{j=1}^{i-1} w_j ReLU(g_j(x))) = T + \sum_{j=1}^{i-1} w_j ReLU(g_j(x)) > 0$$

by doing a linear combination of the two additional nodes of the previous layer. In short, the i^{th} layer up to $i \leq n$ will compute the function

$$h^i(x) = (x_1, \cdots, x_d, ReLU(g_i(x)), ReLU(T + \sum_{j=1}^{i-1} w_j ReLU(g_j(x)))).$$

The final layer then computes

$$\begin{aligned} h^{n+1}(x) &= ReLU[w_n ReLU(g_n(x)) + ReLU(T + \sum_{j=1}^{n-1} w_j ReLU(g_j(x))) - T + b] \\ &= ReLU(b + \sum_{i=1}^{n} w_i ReLU(g_i(x))) \\ &= f(x), \end{aligned}$$

which completes the proof by using n hidden layers, each of width $d + 2$. □

We now move onto two stronger and more fundamental results about ReLU networks:

Definition 3.19. A function $f : \mathbb{R}^n \to \mathbb{R}$ is called affine if it is of the form:

$$f(x) = a \cdot x + b$$

for some $a \in \mathbb{R}^n$ and $b \in \mathbb{R}$.

We may also wish to extrapolate this concept and define:

Definition 3.20. A function $f : \mathbb{R}^n \to \mathbb{R}^m$ is called affine if all its components $f_i : \mathbb{R}^n \to \mathbb{R}$ are affine according to the definition above.

Lemma 3.21. *Let $f : \mathbb{R}^d \to \mathbb{R}$ be the function calculated by an arbitrary neural network with ReLU activation. Then there exist affine functions $g_1, ..., g_N$ and $h_1, ..., h_M$ such that f is of the form*

$$f(x) = g(x) - h(x) \quad with \quad g(x) = \max_{\alpha<N} g_\alpha(x) \quad h(x) = \max_{\beta<M} h_\beta(x). \tag{16}$$

Lemma 3.22. *In addition to that, any function of the form represented above can be calculated exactly by a ReLU network of width $(d + 2)$ and depth $(\max\{M, N\} + 1)$. There also exists a network of width $(d + 1)$ and depth $(M + N)$ with the same property.*

These two lemmas create a nice "if and only if" relation, in which it is shown that all functions created by ReLU feedforward networks are of a specific form, and that all functions of that form can be represented by these networks with bounded width and depth.

In order to prove these theorems, we must first develop some machinery involving affine functions and the $\max\{\cdot\}$ function.

Definition 3.23. Let A, B be sets of real numbers,and x be a real number. Define

$$A + x := \{a + x \mid a \in A\}, \qquad A + B := \{a + b \mid a \in A, b \in B\}, \qquad xA := \{ax \mid a \in A\}. \tag{17}$$

Lemma 3.24. *Let A, B be finite sets of real numbers, and x be a real number. Then the following properties hold:*

(i) $\max\{A\} + \max\{B\} = \max\{A + B\}$

(ii) $\max\{A, \max\{B\}\} = \max\{A, B\}$

(iii) $\max\{A, \max\{B\} + x\} = \max\{A, B + x\}$

(iv) $\max\{A, -\max\{B\} + x\} = \max\{A + B, x\} - \max\{B\}$

(v) $x \max\{A\} = \begin{cases} \max\{xA\} & \text{if } x \geq 0 \\ -\max\{-xA\} & \text{if } x < 0 \end{cases}$

The proofs of these equalities are not too hard to prove, and, to quote Prof. David Williams: "you can also check for yourself that [these] hold: I wouldn't dream of depriving you of that pleasure". The hardest proof among these is likely that of *(iv)*, so it shall be included.

Proof. (of (iv)) We break this problem into two cases, the first one being: $\max\{A, -\max\{B\}+x\} = a \in A$. It follows that

$$a > x - \max\{B\} \rightarrow a + \max\{B\} > x. \tag{18}$$

Then on the right side of *(iv)* we have

$$\max\{A+B, x\} - \max\{B\} = \max\{A\} + \max\{B\} - \max\{B\} = a, \tag{19}$$

which is what we hoped for. The second case, then, is: $\max\{A, -\max\{B\}+x\} = -b + x$ for some $b \in B$. Similarly, it follows that

$$-b + x > \max\{A\} \rightarrow x > \max\{A\} + b = \max\{A\} + \max\{B\} = \max\{A+B\}. \tag{20}$$

Once again, on the right side if *(iv)* we have:

$$\max\{A+B, x\} - \max\{B\} = x - b. \tag{21}$$

□

We now move onto affine functions.

Lemma 3.25. *Let $f, g : \mathbb{R}^n \rightarrow \mathbb{R}$ be affine functions, and α be a real number. Then the following properties hold:*

(i) $(f+g)(x)$ *is affine.*

(ii) $\alpha f(x)$ *is affine.*

(iii) $f(x) + \alpha$ *is affine.*

Once again, proving these properties is not complicated, and "I wouldn't dream of depriving you of that pleasure".

We now finally move onto proving Lemma 3.21.

Proof of Lemma 3.21. We want to show that functions generated by ReLU feedforward networks are of the form in (16). We shall do this using induction on the number of hidden layers of the network.

Base Case: (input layer and output layer are directly connected) Let $a_i \in \mathbb{R}$ be the weight connecting each i^{th} input note to the output node, and $b \in \mathbb{R}$ be the bias. Furthermore, define $a = \sum_{i=1}^{n} a_i e_i$, where e_i are the standard basis vectors. Then the function calculated by the network is

$$f(x) = ReLU(a \cdot x + b) = \max\{0, a \cdot x + b\},$$

which is, indeed, of the form in (16). ✓

Inductive Step: Assume that for a network the values stored in the layer immediately preceding the output node are all calculated by functions of the form in (16). More formally, assume the layer immediately preceding the output node has k nodes, and that for the i^{th} node in this layer, the function calculated in it is of the form

$$f^i(x) = g^i(x) - h^i(x) \quad \text{with} \quad g^i(x) = \max_{\alpha < N^i}\{g^i_\alpha(x)\} \quad h^i(x) = \max_{\beta < M^i}\{h^i_\beta(x)\}.$$

Furthermore, let $w_i \in \mathbb{R}$ be the weight connecting each i^{th} node of the last hidden layer to the output node, and $b \in \mathbb{R}$ be the bias. Then the function calculated by the network is

$$\begin{aligned}
f(x) &= ReLU(b + \sum_{i=1}^{k} w_i f^i(x)) \\
&= \max\{0, b + \sum_{i=1}^{k} w_i [g^i(x) - h^i(x)]\} \\
&= \max\{0, b + \sum_{i=1}^{k} w_i [\max_{\alpha < N^i}\{g^i_\alpha(x)\} - \max_{\beta < M^i}\{h^i_\beta(x)\}]\} \\
&= \max\{0, b + \sum_{i=1}^{k} [w_i \max_{\alpha < N^i}\{g^i_\alpha(x)\} - w_i \max_{\beta < M^i}\{h^i_\beta(x)\}]\}.
\end{aligned}$$

Let us look closer into the sum in the previous expression. By Lemma 3.24, if $w_i \geq 0$, we can bring the w_i 'inside' the brackets, while if $w_i < 0$ we must multiply the term and the argument of the max function by -1 in order to do so. Define the following sets

$$A := \{i \in \{1, ..., k\} | w_i \geq 0\}, \qquad B := \{i \in \{1, ..., k\} | w_i < 0\}. \tag{22}$$

Then we have that

$$\begin{aligned}\sum_{i=1}^{k}(w_i \max_{\alpha<N^i}\{g_\alpha^i(x)\} - w_i \max_{\beta<M^i}\{h_\beta^i(x)\}) = &\sum_{i\in A}\max_{\alpha<N^i}\{w_i g_\alpha^i(x)\} - \sum_{i\in B}\max_{\alpha<N^i}\{-w_i g_\alpha^i(x)\} \\ &- \sum_{i\in A}\max_{\beta<M^i}\{w_i h_\alpha^i(x)\} + \sum_{i\in B}\max_{\beta<M^i}\{-w_i h_\alpha^i(x)\}.\end{aligned}$$

It should be noted that, by Lemma 3.25, all the $w_i g_\alpha^i(x)$ and $w_i h_\alpha^i(x)$, or linear combinations of them are still affine functions. By repeatedly applying Lemma 3.24(i), we therefore conclude that

$$\sum_{i\in A}\max_{\alpha<N^i}\{w_i g_\alpha^i(x)\} + \sum_{i\in B}\max_{\beta<N^i}\{-w_i h_\alpha^i(x)\} = \max\{F_1\}$$

and

$$\sum_{i\in B}\max_{\alpha<N^i}\{-w_i g_\alpha^i(x)\} + \sum_{i\in A}\max_{\beta<N^i}\{w_i h_\alpha^i(x)\} = \max\{F_2\},$$

where F_1, F_2 are finite families of affine functions. We then use this fact and Lemma 3.24 to finish the proof.

$$\begin{aligned}f(x) &= \max\{0, b + \sum_{i=1}^{k}(w_i \max_{\alpha<N^i}\{g_\alpha^i(x)\} - w_i \max_{\beta<M^i}\{h_\beta^i(x)\})\} \\ &= \max\{0, b + \max\{F_1\} - \max\{F_2\}\} \\ &= \max\{0, \max\{b + F_1\} - \max\{F_2\}\} \\ &= \max\{F_2, b + F_1\} - \max\{F_2\}\end{aligned}$$

□

Now in order to prove Lemma 3.22, we will first prove the following result, which is included in [Han17]:

Lemma 3.26. *Let $d \in \mathbb{Z}_+$, $T : \mathbb{R}_+^d \to \mathbb{R}$ be an arbitrary function and $L : \mathbb{R}^d \to \mathbb{R}$ be affine. Define the affine function $A : \mathbb{R}^{d+1} \to \mathbb{R}^{d+1}$ by*

$$A(x, y) := (x, L(x) + y) \tag{23}$$

where $x \in \mathbb{R}_+^d$ and $y \in \mathbb{R}$. Then we have that

$$(A \circ ReLU \circ A^{-1})(x, T(x)) = (x, \max\{L(x), T(x)\}). \tag{24}$$

Proof.

$$\begin{aligned}(A \circ ReLU)(A^{-1}(x, T(x))) &= (A \circ ReLU)(x, T(x) - L(x)) \\ &= A(x, \max\{0, T(x) - L(x)\}) \\ &= (x, \max\{0, T(x) - L(x)\} + L(x)) \\ &= (x, \max\{0 + L(x), T(x) - L(x) + L(x)\}) \\ &= (x, \max\{L(x), T(x)\})\end{aligned}$$

□

In a neural network setting, this lemma implies that we can calculate the maximum of two numbers using a node, while copying all the other nodes to the following layer.We will use repeated iterations of this lemma in order to prove our desired result.

Proof of Lemma 3.22. First consider f of the form

$$f(x) = \max_{\alpha \leq N}\{g_\alpha(x)\}.$$

Using the notation from the previous lemma, for any $\alpha \in \{1, \dots, N\}$ define

$$A_\alpha(x, y) := (x, y + g_\alpha(x)).$$

And for any $\alpha \in \{2, \dots, N\}$, set

$$H_\alpha(x, y) := (A_\alpha^{-1} \circ A_{a-1})(x, y) = (x, y + g_{\alpha-1}(x) - g_\alpha(x)).$$

These are still affine functions. Additionally, define $H_1 : \mathbb{R}^d \to \mathbb{R}^{d+1}$ and $H_{N+1} : \mathbb{R}^{d+1} \to \mathbb{R}$ by

$$H_1(x) := (x, -g_1(x)) \qquad H_{N+1}(x, y) := (y + g_N(x)).$$

Then, by using H_i as the affine functions that connect the i^{th} layer to the next one (the first one being the input layer), we use the lemma to conclude that the network will compute $\max_{\alpha \leq N}\{g_\alpha(x)\}$ exactly using $d + 1$ nodes in each hidden layer, and N hidden layers. That is, the function F generated by the network is

$$\begin{aligned} F(x) &= (H_{N+1} \circ ReLU \circ H_N \circ \dots \circ H_2 \circ ReLU \circ H_1)(x) \\ &= (H_{N+1} \circ ReLU \circ A_{N-1} \circ A_N^{-1} \circ ReLU \circ \dots \circ ReLU \circ A_1 \circ A_2^{-1} \circ ReLU \circ H_1)(x) \\ &= \max_{\alpha \leq N}\{g_\alpha(x)\} \\ &= f(x). \end{aligned}$$

To finish the proof, we consider the general case:

$$f(x) = \max_{\alpha \leq N}\{g_\alpha(x)\} - \max_{\beta \leq M}\{h_\beta(x)\}.$$

For this, we use an additional node in each hidden layer to compute $\max_{\beta \leq M}\{h_\beta(x)\}$ in parallel with the computations used to obtain $\max_{\alpha \leq N}\{g_\alpha(x)\}$. At the end of this process, we will have used $d + 2$ nodes in each hidden layer, and $\max\{M, N\}$ hidden layers to output two nodes: one with the value of $g(x)$, the other with the value of $h(x)$. We thus use a final layer to compute their difference and we are done with the proof. The final network has width $d + 2$ and depth $(\max\{M, N\} + 1)$.

Alternatively, we can use a design which, once again, uses d nodes in each layer to copy the original inputs, and only one additional node in each layer. Using this strategy, we first use N layers to calculate the value of $G := g(x)$. Since G is just a real number, calculating $\max_{\beta \leq M}\{h_\beta(x) + G\}$ requires the exact same strategy as just calculating $\max_{\beta \leq M}\{h_\beta(x)\}$, since an affine function plus a constant is still an affine function. As a result, we calculate the two functions in *series* using first N layers to calculate G and then M more layers to calculate the final function $f = g - h$.

□

We can also generalize this result for functions with other output dimensions.

Theorem 3.27. *Any function $f : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$ of the form $f(x) = (f_1(x), \dots, f_{d_{out}})$ with each f_i of the form:*

$$f^i(x) = g^i(x) - h^i(x) \quad \text{with} \quad g^i(x) = \max_{\alpha < N^i}\{g^i_\alpha(x)\} \quad h^i(x) = \max_{\beta < M^i}\{h^i_\beta(x)\} \tag{25}$$

can be calculated by a ReLU network of width $d_{in} + d_{out}$ and depth $\max_{i=1,\dots,d_{out}}(N^i + M^i)$.

Proof. Following the same setup used in the proof of the previous theorem, we use the strategy of keeping d nodes in each layer reserved to pass along the original values imputed in the network, and use an additional node for the computation of each f_i. This way, the components of f are calculated in parallel until the longest (meaning the one with the largest value for $(M_i + N_i)$) finishes computing. The resulting network has width $d_{in} + d_{out}$ and depth $\max_{i=(1,\dots,d_{out})}(N_i + M_i)$. □

As in the previous theorem, it is also possible to decrease the depth of the network to $\max_{i=(1,\dots,d_{out})}(N_i, M_i)$ at the cost of increasing the width to $d_{in} + 2d_{out}$. Once again, this does not change the computational cost of the network if it is of the form described in the proof.

Consider the following concept:

Definition 3.28. Let $w_{min}^{d_in,d_out}$ be defined as

$$w_{min}^{d_{in},d_{out}} := \min\{w | \text{for any} f : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}, \epsilon > 0 \exists \text{ a network } \mathcal{N} \text{ with width}(\mathcal{N}) = w \text{ such that } |f_{\mathcal{N}} - f| < \epsilon\}.$$

This leads to a rephrasing of the previous theorem:

Corollary 3.29. *For any $d_{in}, d_{out} \geq 1$ we have that*

$$w_{min}^{d_{in},d_{out}} \leq d_{in} + d_{out}. \tag{26}$$

An interesting theorem from [HS17] provides a lower bound to the situation above:

Theorem 3.30. *For any $d_{in}, d_{out} \geq 1$ we have that*

$$w_{min}^{d_{in},d_{out}} \geq d_{in} + 1. \tag{27}$$

In other words, if a ReLU network has all its hidden layers with at most d_{in} nodes, then there exists some function $f : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$ and some $\epsilon > 0$ such that no matter how the weights and biases are adjusted, $|f_{\mathcal{N}} - f| > \epsilon$. This has very direct consequences to the implementation of ReLU networks: if one wishes to create the most versatile network possible, then at least one of its layers must have $d_{in} + 1$ nodes. However, since the input space of the functions approximated by neural networks tend to be very big, this at times can only be achieved at a very large computational cost.

4 Implementation and Case Study of Efficiency

For the final section of this expository paper, we include a case study of the most classic example of neural network implementation: the MNIST database of handwritten digits. The MNIST database is composed of 70,000 pictures, each of which is made of a 28 by 28 grid of black and white pixels. Each pixel stores a value between 0 and 1, corresponding to the shade of that region, with 0 indicating a completely white square, and 1 indicating a completely black square[2].

Our objective, then, is to create a neural network which is able to classify what type of digit is represented in the picture. We may understand this as trying to approximate some function $f : I_{784} \to I_{10}$, which receives 784 numbers between 0 and 1 as inputs, and returns 10 numbers, each corresponding to the likelihood of the picture relating to a certain number. After the network outputs these 10 numbers, we take the digit corresponding to the largest of these values as the classification of our network. This is a great application of neural networks, as it would be most likely impossible to write down f explicitly.

4.1 Procedure

In order to experiment around some parameters such as depth, width and learning rate of the network, we use a Python 3.6 implementation of the algorithm presented at the end of the first chapter of [Nie18] (again, available at: `http://neuralnetworksanddeeplearning.com/`). In this implementation, additional techniques which have not been explored in this paper have been employed, most importantly, an instance of backpropagation called *Stochastic Gradient Descent*. This technique aims to ensure that the training method is carried out in a randomly uniform way across the training data, thus making it less likely that the network will specialize in a specific case (say, pictures of the digit 0) and perform badly on others (say, any other digit).

In summary, we divide the 70,000 pictures into a group of 10,000, called the *test set*, and another of 60,000, called the *training set*. We also divide the training process into what are called *epochs*. In each epoch, we randomly select a given number (called *mini-batch size*) of pictures from the training set. We then go through the process of backpropagation for these pictures and finally update the network by adding up all the contributions of the individual pictures to the gradient of the cost function, thus ending this epoch. We repeat this process a given number of times until our training is done. After this step, we test the network on all the 10,000 pictures we separated in the test set without updating the network for these pictures. This separation is made to ensure that our network is not *memorizing* the desired inputs of the training set but rather is able to generalize to results outside of this set. By keeping the two sets separated, we make sure that our test is independent of the training data used to shape the network.

For all networks in this section, the mini-batch size used was composed of 10 training images.

[2]The original database was composed of pictures which were 20 by 20 pixels and only allowed values of shades to be either 0 or 1. However, when re-scaling the pictures to be 28 by 28 pixels, different shades were allowed, as to better preserve the information contained in the pictures.

4.2 Comparison Results

4.2.1 Learning Rate

In order to test how the learning rate of a network affects its accuracy, we used a structure of two hidden layers with 30 nodes each, and varied the learning rate in regular steps. The results obtained are presented in Figure 2, where each point represents a network trained for 20 epochs.

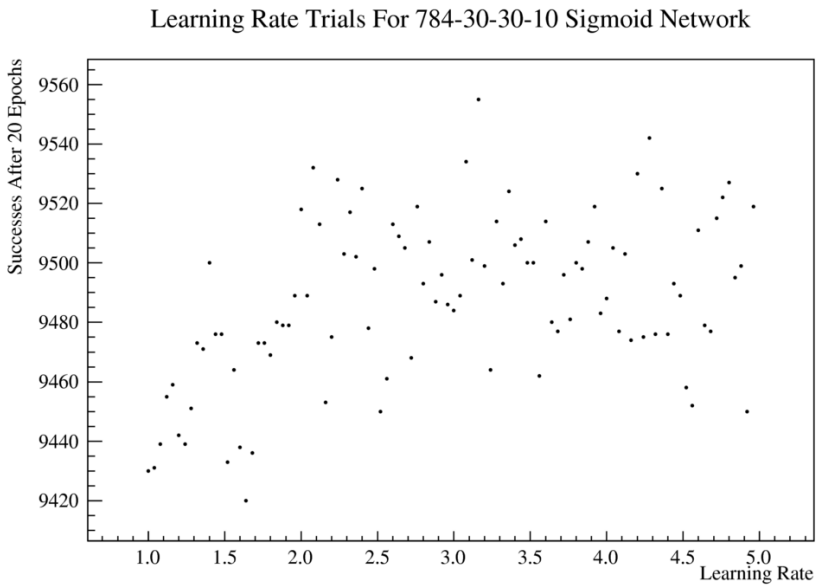

Figure 2: Learning rate trials using a network with two hidden layers of 30 nodes each. The number of successes is based on a testing sample of 10000 pictures.

This graph can be roughly separated into two regions: low learning rates; and larger learning rates. In this first region, located at the left side of the graph, we can notice an increase in accuracy as the learning rate goes up. However, after a certain threshold, we reach the second region of the graph, where points appear more spread out, and accuracy seems to have stagnated when seen in relation to the learning rate.

More will be said about the influence of Learning Rates to the accuracy of the network when we discuss depth.

4.2.2 Width

In an analogous fashion to the previous section, we fix depth and learning rate, and vary the width of the hidden layers in order to examine what relationship this parameter has with the final network accuracy. In Figure 3, results are shown for networks of 2 hidden layers and learning rate of 3. These results show a very clear relationship between accuracy and width of the network, where increasing the number of nodes in each layer increases the number of successes. In the context of 'thinner' networks, adding nodes to the layers makes a very big difference, but this gain diminished as the network grows bigger.

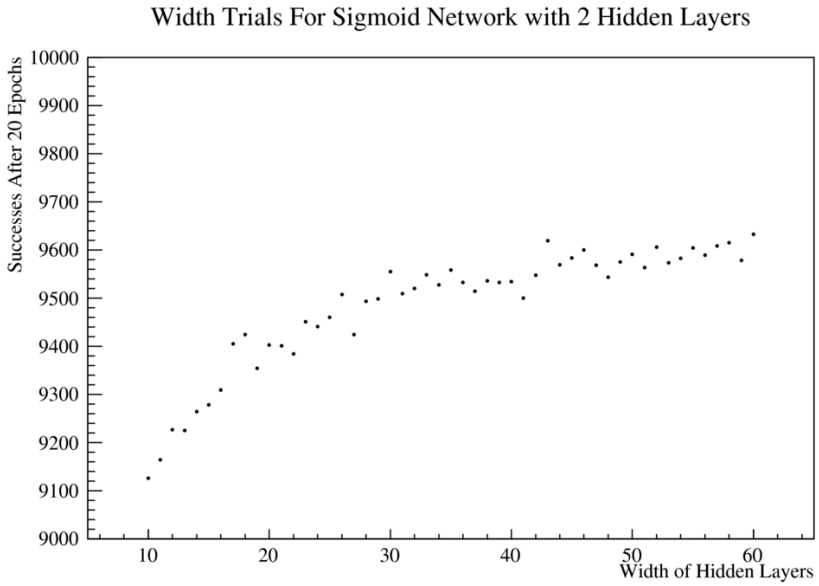

Figure 3: Width trials using a network with two hidden layers with the same number of (varied) widths and learning rate of 3. The number of successes is based on a testing sample of 10000 pictures.

At first sight, this might lead the reader to conclude that in any context having wider networks means having better results. However, when data from Figure 3 is extended to include considerably wider networks, results are less encouraging.

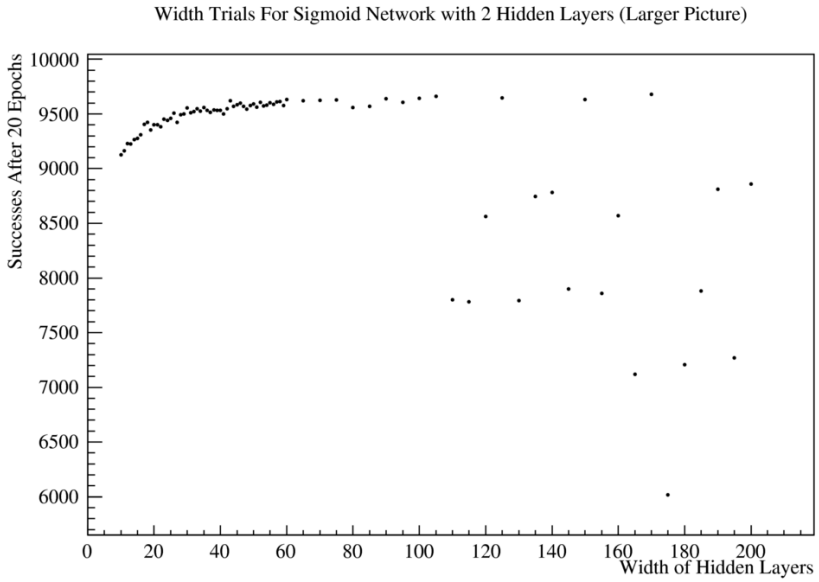

Figure 4: Width trials using a network with two hidden layers with the same number of (varied) widths and learning rate of 3. The number of successes is based on a testing sample of 10000 pictures.

In Figure 4, the accuracy of the networks does increase as a function of width, up to roughly about 100 hidden layers. After this threshold, network accuracy varies significantly, in most cases for the worse. This is because a wider network has many more weights and biases that need to be adjusted, so the learning process happens more slowly. In this specific case of networks with two hidden layers of width n, 784 inputs and 10 outputs, the number of weights is $784n+n^2+10n = n^2+794n$ and the number of biases is $n+n+10 = 2n+10$ which means that the total number of parameters that need to be adjusted is $n^2+796n+10$. Because of this squared dependency on n, larger networks will require many more epochs of training in order to achieve their full potential. Not only that, but each epoch requires much more computational power in larger networks. However, the large number of parameters also means that the network is capable of expressing a larger class

of functions, which is indeed shown when considering networks trained over more epochs. This is shown in Figure 5, which compares networks with 50, 100 and 200 nodes in each hidden layer across different epochs.

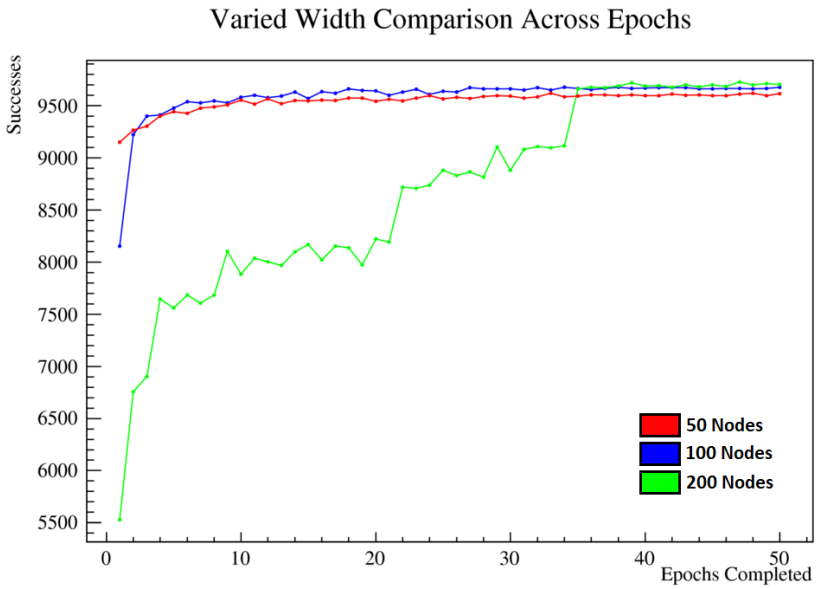

Figure 5: Trials using sigmoid networks with two hidden layers and learning rate of 3 across 50 epochs. The number of successes is based on a testing sample of 10000 pictures.

As expected, larger networks require more training epochs in order to achieve their full potential, but when this is obtained, they indeed perform better than smaller networks. In this specific case, however, the gain is noticeable, but not overwhelming, while the required time to train larger networks is, indeed much larger. A zoomed in version of Figure 5 is presented in Figure 6 for readability purposes.

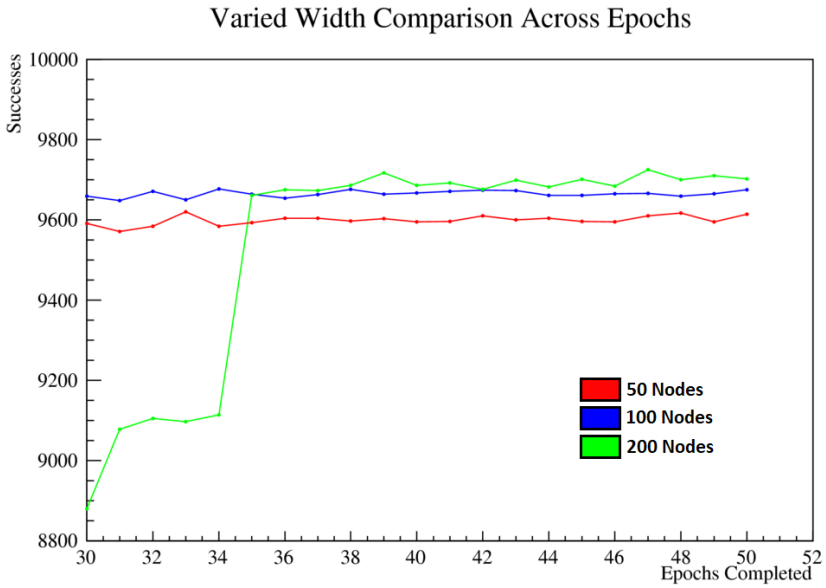

Figure 6: Trials using sigmoid networks with two hidden layers and learning rate of 3 across 50 epochs. The number of successes is based on a testing sample of 10000 pictures.

4.2.3 Depth

"Deep Learning" is a term which became popular in machine learning over the past few years, with many news articles praising successful applications of the method. This technique simply means that many layers are being used in a neural network (although what qualifies as "many" is not always well defined). Although

this method has indeed been incredibly successful in many important applications, the reader should not be mislead into believing that more layers always means better accuracy. For instance, consider the results exhibited in Figure 7, where networks with different depth were compared.

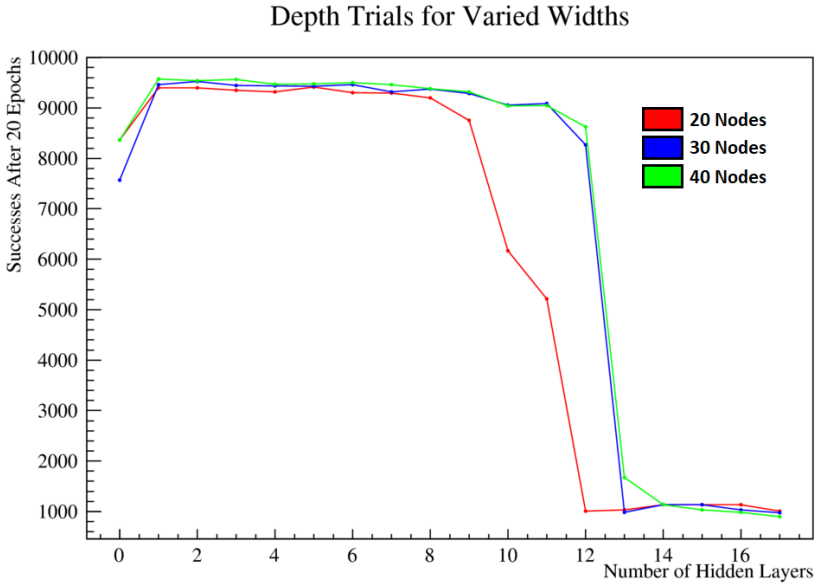

Figure 7: Networks used had a learning rate of 3 and sigmoid activation function. The number of successes is based on a testing sample of 10000 pictures.

After a certain point, having more layers in the network dramatically decreases its accuracy, to a point where it converges to 10%, which is equivalent to just randomly guessing what digit is in the picture! A reasonable thought to have after reading the previous section is to conclude that this is because the network is larger, and therefore requires more epochs of training in order to achieve its true potential. However, further examination shows that even after 40 epochs, the network does not show any sign of improvement. However, observe in Figure 8 what happens when we lower the learning rate of the network.

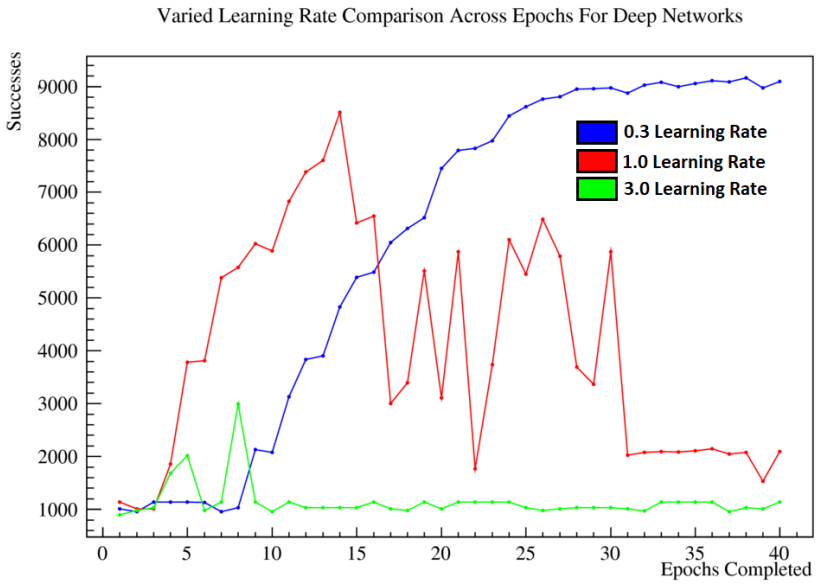

Figure 8: Trials for sigmoid networks of 16 hidden layers with 20 nodes each. The number of successes is based on a testing sample of 10000 pictures.

Each curve presented in Figure 8 relates to a different region of influence of the learning rate. In the first one, represented by 0.3, the learning rate is appropriate, and, given enough training epochs, results in a

successful network. In the second one, represented by 3.0, the learning rate is so big that the steps taken by the gradient descent technique completely skip over any minimum of the cost function, turning the network's output into random guesses. The final one, represented by 1.0, is an intermediary region, where steps are small enough that the outcome of the network is definitely better than random guesses, but still too big, as to make the network oscillate intensely across one or more minimums of the cost function, resulting in unreliability.

Why does the deeper network require a smaller learning rate in order to function properly when compared to shallower networks? The answer relies on the different types of operations carried out when shaping the network's function: in shallow but wide networks, the primary operations influencing the outcome are affine combinations, which are determined by the weights and biases across layers; in deep but thin networks function composition is the key, which means that each individual weight (specially closer to the input layer) has a very large influence on the outcome of the network. As a result, in deeper networks we need smaller increments to the gradient descent in order to finely tune it to the desired dataset.

Acknowledgements

This paper was only made possible thanks to the University of Chicago's Mathematics Department and the Research Experience for Undergraduates organized by Prof. Peter May, who year after year excels at coordinating this endeavor. The author would also like to thank his mentor, Brian Chung, whose help was crucial over the past few months and who contributed immensely to the editing of this paper.

References

[Bre10] H. Brezis. *Functional Analysis, Sobolev Spaces and Partial Differential Equations*. Universitext. Springer New York, 2010.

[Cyb89] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, December 1989.

[Han17] Boris Hanin. Universal function approximation by deep neural nets with bounded width and relu activations, 2017.

[HS17] Boris Hanin and Mark Sellke. Approximating continuous functions by relu nets of minimal width, 2017.

[LLPS93] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6:861–867, 1993.

[Nie18] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2018.

Appendix A Data

Since the algorithm employed used the stochastic gradient descent technique and is, therefore, partially random, the data displayed in the graphs of Section 4 is detailed below for the sake of completeness. In order to retain compactness, the following abbreviations are used: L.R. (Learning Rate); C.G. (Correct Guesses); E.C. (Epochs Completed); Wd. (Width); Dp. (Depth).

Table 1: Data for Figure 2

L.R.	C.G.
1.00	9430
1.04	9431
1.08	9439
1.12	9455
1.16	9459
1.20	9442
1.24	9439
1.28	9451
1.32	9473
1.36	9471
1.40	9500
1.44	9476
1.48	9476
1.52	9433
1.56	9464
1.60	9438
1.64	9420
1.68	9436
1.72	9473
1.76	9473

L.R.	C.G.
1.80	9469
1.84	9480
1.88	9479
1.92	9479
1.96	9489
2.00	9518
2.04	9489
2.08	9532
2.12	9513
2.16	9453
2.20	9475
2.24	9528
2.28	9503
2.32	9517
2.36	9502
2.40	9525
2.44	9478
2.48	9498
2.52	9450
2.56	9461

L.R.	C.G.
2.60	9513
2.64	9509
2.68	9505
2.72	9468
2.76	9519
2.80	9493
2.84	9507
2.88	9487
2.92	9496
2.96	9486
3.00	9484
3.04	9489
3.08	9534
3.12	9501
3.16	9555
3.20	9499
3.24	9464
3.28	9514
3.32	9493
3.36	9524

L.R.	C.G.
3.40	9506
3.44	9508
3.48	9500
3.52	9500
3.56	9462
3.60	9514
3.64	9480
3.68	9477
3.72	9496
3.76	9481
3.80	9500
3.84	9498
3.88	9507
3.92	9519
3.96	9483
4.00	9488
4.04	9505
4.08	9477
4.12	9503
4.16	9474

L.R.	C.G.
4.20	9530
4.24	9475
4.28	9542
4.32	9476
4.36	9525
4.40	9476
4.44	9493
4.48	9489
4.52	9458
4.56	9452
4.60	9511
4.64	9479
4.68	9477
4.72	9515
4.76	9522
4.80	9527
4.84	9495
4.88	9499
4.92	9450
4.96	9519

Table 2: Data for figures 3 and 4

Wd.	C.G.
10	9126
11	9164
12	9227
13	9225
14	9264
15	9278
16	9309
17	9405
18	9424
19	9354
20	9402
21	9401
22	9384
23	9451
24	9441
25	9460

Wd.	C.G.
26	9507
27	9424
28	9493
29	9498
30	9555
31	9509
32	9520
33	9548
34	9527
35	9558
36	9532
37	9514
38	9536
39	9532
40	9534
41	9500

Wd.	C.G.
42	9547
43	9619
44	9569
45	9583
46	9600
47	9568
48	9543
49	9575
50	9591
51	9563
52	9606
53	9573
54	9582
55	9604
56	9589
57	9608

Wd.	C.G.
58	9615
59	9578
60	9632
65	9620
70	9625
75	9628
80	9558
85	9571
90	9638
95	9607
100	9641
105	9660
110	7799
115	7782
120	8561
125	9647

Wd.	C.G.
130	7793
135	8746
140	8783
145	7901
150	9633
155	7859
160	8569
165	7119
170	9680
175	6017
180	7208
185	7880
190	8811
195	7268
200	8860

Table 3: Data for figures 5 and 6

Wd.	E.C.	C.G.
50	1	9148
50	2	9264
50	3	9303
50	4	9399
50	5	9443
50	6	9426
50	7	9478
50	8	9486
50	9	9507
50	10	9554
50	11	9513
50	12	9566
50	13	9520
50	14	9551
50	15	9546
50	16	9552
50	17	9550
50	18	9572
50	19	9573
50	20	9543
50	21	9562
50	22	9546
50	23	9574
50	24	9597
50	25	9565
50	26	9581
50	27	9568
50	28	9586
50	29	9597
50	30	9591
50	31	9571
50	32	9584
50	33	9620
50	34	9584
50	35	9593
50	36	9604
50	37	9604
50	38	9597
50	39	9603
50	40	9595
50	41	9596
50	42	9610
50	43	9600
50	44	9604
50	45	9596
50	46	9595
50	47	9610
50	48	9617
50	49	9595
50	50	9614

Wd.	E.C.	C.G.
100	1	8151
100	2	9223
100	3	9399
100	4	9412
100	5	9476
100	6	9539
100	7	9526
100	8	9545
100	9	9528
100	10	9579
100	11	9601
100	12	9575
100	13	9592
100	14	9630
100	15	9570
100	16	9635
100	17	9618
100	18	9660
100	19	9644
100	20	9640
100	21	9599
100	22	9630
100	23	9657
100	24	9608
100	25	9637
100	26	9630
100	27	9672
100	28	9660
100	29	9662
100	30	9659
100	31	9648
100	32	9671
100	33	9650
100	34	9677
100	35	9664
100	36	9654
100	37	9663
100	38	9676
100	39	9664
100	40	9667
100	41	9671
100	42	9674
100	43	9673
100	44	9661
100	45	9661
100	46	9665
100	47	9666
100	48	9659
100	49	9665
100	50	9675

Wd.	E.C.	C.G.
200	1	5527
200	2	6757
200	3	6904
200	4	7645
200	5	7559
200	6	7683
200	7	7605
200	8	7685
200	9	8102
200	10	7882
200	11	8036
200	12	8003
200	13	7968
200	14	8098
200	15	8167
200	16	8022
200	17	8153
200	18	8138
200	19	7973
200	20	8222
200	21	8193
200	22	8718
200	23	8705
200	24	8739
200	25	8878
200	26	8831
200	27	8865
200	28	8815
200	29	9102
200	30	8880
200	31	9078
200	32	9105
200	33	9097
200	34	9114
200	35	9661
200	36	9675
200	37	9673
200	38	9686
200	39	9717
200	40	9686
200	41	9692
200	42	9676
200	43	9699
200	44	9682
200	45	9701
200	46	9684
200	47	9725
200	48	9700
200	49	9710
200	50	9702

Table 4: Data for Figure 7

Wd.	Dp.	C.G.
20	1	8363
20	2	9400
20	3	9400
20	4	9349
20	5	9318
20	6	9414
20	7	9306
20	8	9291
20	9	9198
20	10	8750
20	11	6172
20	12	5213
20	13	1009
20	14	1032
20	15	1135
20	16	1135
20	17	1135
20	18	1010

Wd.	Dp.	C.G.
30	1	7569
30	2	9461
30	3	9521
30	4	9447
30	5	9435
30	6	9430
30	7	9463
30	8	9315
30	9	9376
30	10	9286
30	11	9059
30	12	9087
30	13	8272
30	14	982
30	15	1135
30	16	1135
30	17	1032
30	18	974

Wd.	Dp.	C.G.
40	1	8367
40	2	9570
40	3	9543
40	4	9562
40	5	9468
40	6	9476
40	7	9502
40	8	9460
40	9	9380
40	10	9316
40	11	9041
40	12	9049
40	13	8630
40	14	1677
40	15	1135
40	16	1028
40	17	980
40	18	892

Table 5: Data for Figure 8

L.R.	E.C.	C.G.
0.3	1	1010
0.3	2	958
0.3	3	1135
0.3	4	1135
0.3	5	1135
0.3	6	1132
0.3	7	958
0.3	8	1028
0.3	9	2134
0.3	10	2076
0.3	11	3134
0.3	12	3838
0.3	13	3902
0.3	14	4826
0.3	15	5390
0.3	16	5486
0.3	17	6050
0.3	18	6315
0.3	19	6517
0.3	20	7450
0.3	21	7792
0.3	22	7831
0.3	23	7970
0.3	24	8440
0.3	25	8620
0.3	26	8765
0.3	27	8808
0.3	28	8951
0.3	29	8959
0.3	30	8971

L.R.	E.C.	C.G.
0.3	31	8879
0.3	32	9026
0.3	33	9077
0.3	34	8998
0.3	35	9056
0.3	36	9113
0.3	37	9091
0.3	38	9161
0.3	39	8974
0.3	40	9099
1.0	1	1135
1.0	2	1009
1.0	3	1009
1.0	4	1857
1.0	5	3780
1.0	6	3812
1.0	7	5384
1.0	8	5581
1.0	9	6028
1.0	10	5893
1.0	11	6833
1.0	12	7384
1.0	13	7601
1.0	14	8513
1.0	15	6418
1.0	16	6551
1.0	17	3004
1.0	18	3396
1.0	19	5510
1.0	20	3105

L.R.	E.C.	C.G.
1.0	21	5877
1.0	22	1770
1.0	23	3740
1.0	24	6104
1.0	25	5450
1.0	26	6489
1.0	27	5792
1.0	28	3693
1.0	29	3368
1.0	30	5872
1.0	31	2021
1.0	32	2080
1.0	33	2091
1.0	34	2082
1.0	35	2110
1.0	36	2145
1.0	37	2046
1.0	38	2079
1.0	39	1529
1.0	40	2090
3.0	1	892
3.0	2	980
3.0	3	1028
3.0	4	1684
3.0	5	2017
3.0	6	975
3.0	7	1135
3.0	8	2998
3.0	9	1135
3.0	10	958

L.R.	E.C.	C.G.
3.0	11	1135
3.0	12	1028
3.0	13	1028
3.0	14	1028
3.0	15	1028
3.0	16	1135
3.0	17	1010
3.0	18	982
3.0	19	1135
3.0	20	1010
3.0	21	1135
3.0	22	1135
3.0	23	1135
3.0	24	1135
3.0	25	1028
3.0	26	980
3.0	27	1010
3.0	28	1028
3.0	29	1032
3.0	30	1028
3.0	31	1009
3.0	32	974
3.0	33	1135
3.0	34	1135
3.0	35	1135
3.0	36	1135
3.0	37	958
3.0	38	1028
3.0	39	1009
3.0	40	1135