Exploring the Forward-Forward Algorithm

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in

Data Science

By:

Jared Leonard

Project Advisor:

Randy Paffenroth

December 15, 2023

Abstract

This report explores using the Foward-Foward (FF) algorithm to train neural networks. Unlike traditional training methods that use a forward and backward pass, the FF algorithm uses two forward passes with opposite objectives. The findings of this report lay the groundwork for feature work on the FF algorithm and propose a new method to control the influence individual layers have on the network's performance.

Acknowledgements

I want to thank my project advisor, Professor Randy Paffenroth, whose expertise and guidance has been instrumental in completing this project. His mentorship has deepened my understanding of complex topics and played an important role in developing my research skills. This experience has been extremely valuable, providing me with many helpful lessons that have contributed to my personal and professional development.

Contents

List of Tables

List of Figures

1 Introduction

Since its introduction in the late 1980s, gradient descent using backpropagation has emerged as one of the predominant methods for training neural networks [2]. It operates in two phases: the forward pass and the backward pass. In the forward pass, input data is fed through the network, layer by layer, each transforming the data until an output is produced. This output is then compared against the desired outcome, and the discrepancy between the two indicates the error. The backward pass then propagates the error backward through the network, updating the network's parameters with the goal of minimizing the error [3]. This systematic approach of adjusting parameters to reduce error has enabled neural networks to be trained to perform some remarkable tasks. However, despite its effectiveness, backpropagation has limitations.

One critique of using gradient descent with backpropagation is that its methodology does not align with the human brain's learning mechanisms [4]. Unlike machine learning models, the human brain constantly processes sensory inputs. It can interpret and learn from a continuous stream of information in real time. In contrast, backpropagation necessitates the network pause momentarily to compute the gradients and then adjust its parameters based on the calculated error. This critique of backpropagation has led many experts to the consensus that it is biologically implausible for the human brain to employ backpropagation as a learning mechanism [5].

Another drawback of gradient descent with backpropagation is its reliance on the forward pass. If a complete understanding of the computations performed during the forward pass is unavailable, backpropagation cannot function effectively. Lacking this essential information makes it impossible to compute the partial derivatives of the loss function, leaving the network without guidance on how to adjust its weights and biases. Furthermore, if a black box exists in the forward pass, backpropagation alone becomes insufficient, necessitating alternative approaches like reinforcement learning, which can introduce other issues [6].

In late 2022, Geoffrey Hinton proposed a new approach to network training, introducing an algorithm he dubbed the Forward-Forward (FF) algorithm [4]. The FF algorithm addresses some of the challenges of backpropagation by implementing a fundamentally different learning mechanism. Instead of the traditional forward and backward passes, FF uses two forward passes. One pass processes positive real data, while the other deals with negative data, which could be self-generated by the network. This approach aims to simplify the learning process, potentially allowing for real-time data processing without storing neural activities or pausing for backpropagation, thus offering a more continuous learning model that may align

more closely with biological learning processes [4].

The ultimate goal of this project is to understand and assess the Forward-Forward (FF) algorithm proposed by Geoffrey Hinton. During the assessment, the aim will be to understand the algorithm's capabilities and identify its strengths, limitations, and suitability for real-world application. Hopefully, this evaluation will help inform the potential of future work on the FF algorithm. Additionally, this project will explore if the FF algorithm can overcome the inherent limitations of backpropagation, offering a more biologically plausible and efficient learning mechanism for neural networks.

2 Background

2.1 Notation and Definitions

Notation	**Description**	**Example**
x	Scalar	3.14
$\mathbf{x}$	Column vector	$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$
$\lvert\mathbf{x}\rvert$	The number of entries in $\mathbf{x}$	$\left\lvert \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \right\rvert = 3$
X	Matrix	$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

Table 1: Notation for scalars, vectors, and matrices.

2.2 Neural Networks

Researchers created neural networks as an attempt to replicate the decision-making process of the human brain, inspired by its structure and functionality. The overarching goal was to enable machines to learn from observational data, much like humans learn from experience [7]. Consider a relationship between two variables, x and y, expressed as $y = f(x) + \epsilon$ where f represents an unknown function and ϵ is a random error term. With a sufficient amount of data that captures the nuances of the relationship between x and y, it is possible to determine the unknown function f that accurately represents the relationship between

the two variables. Once the unknown function f is identified, it enables y to be predicted if x is known. However, for many relationships, the exact function remains unknown primarily due to factors influencing y that are not captured in x[8].

As the relationship between variables becomes increasingly complex, conventional statistical approaches may falter due to their inherent limitations in capturing nonlinear and high-dimensional interactions [9]. In contrast, neural networks thrive in these environments, often outperforming traditional models[10]. However, this superior performance does not imply a fundamental departure from the core principles of statistical modeling; just like conventional statistical modeling, neural networks work to identify functions that encapsulate the relationship between variables. During classification tasks, neural networks aim to discover a function that correctly maps inputs with their respective categories. This mapping can be modeled by the equation, $\hat{y} = f(x;\theta)$ where x represents the input features and $\hat{y}$ is the predicted category [11]. The parameters θ are adjusted throughout the training process to align the network's predictions as closely as possible with the actual output categories. Ultimately, the goal is to align the model with the underlying patterns within the data.

2.2.1 Structure

Fully connected neural networks consist of numerous interconnected nodes, or neurons, organized into layers, with each layer serving as a mathematical function that processes incoming data. Most networks have at least three layers: the input, hidden, and output layers [7]. Data enters the network through the first layer, denoted by $f^{(1)}$, typically called the input layer. This layer acts as a gateway for the data to enter and interact with the network. As the data proceeds, it moves through one or more hidden layers, $f^{(2)}$, that transforms it. These hidden layers typically contain most of the network's neurons and are responsible for the model's ability to learn intricate patterns in the data. Once the data reaches the output layer, $f^{(3)}$, the final layer in the network, it undergoes a final transformation to generate the network's output. In the case of classification tasks, the network's final output typically represents the probability that the input belongs to a given class. Sequentially linking all of the layers creates a neural network, resulting in the function $\hat{\mathbf{y}} = f^{(3)}_{\theta_{(3)}}(f^{(2)}_{\theta_{(2)}}(f^{(1)}_{\theta_{(1)}}(\mathbf{x})))$, where $\mathbf{x}$ is the input (like an image in image recognition) and $\hat{\mathbf{y}}$ is the output or prediction (such as identifying the object in the image). Each $f^{(i)}_{\theta_{(i)}}$ denotes the function performed by the i-th layer of the network, parameterized by $\theta_{(i)}$, which the network learns during training. Figure 1 visually depicts this network architecture.

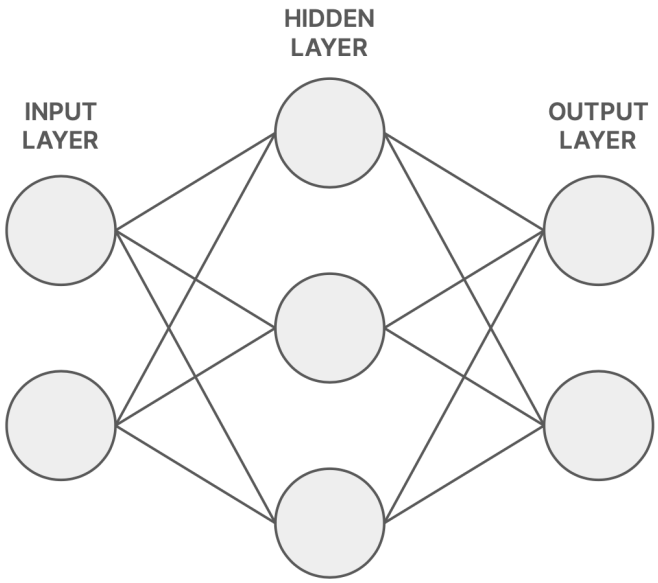


Figure 1: A diagram of a basic fully connected neural network with an input layer containing two neurons, a hidden layer with three neurons, and an output layer with two neurons.

The network depicted in Figure 1 is a fully connected 2-3-2 neural network, where the dots represent neurons, while the connecting lines represent the connections between them. Each neuron in the network contains an activation, which is a mathematical representation of the neuron's current value[11]. Every neuron in a network, apart from those in the input layer, computes its activation based on the outputs from all neurons in the preceding layer. In the input layer, the activation values are derived from external input data, like image pixel values, instead of being calculated. As the network processes an input $\mathbf{x}$, the activations of the neurons are computed dynamically and will vary with each input $\mathbf{x}$. For tasks such as image classification, each neuron in the output layer represents a category $\mathbf{y}$, and when given an input, the activation value of each output neuron represents the likelihood that the image belongs to its corresponding category.

During the training phase, the aim is to adjust the network's parameters θ so that, given an input $\mathbf{x}$, the correct output neuron achieves the greatest activation[11]. To calculate the activations of neurons beyond the input layer, the way in which the neurons are connected must be examined. In a fully connected neural network, every neuron is connected to every neuron in the previous layer. These connections between neurons have an associated weight $\mathbf{w}$ that is adjustable during training. The first step in determining a neuron's activation involves calculating the weighted sum of activations from the preceding layer. This process incorporates the weights associated with each connection, allowing the learned importance of each

input to be factored in. Moreover, every neuron beyond the input layer has a bias term b that can also be adjusted during training. This bias term, an adjustable constant, ensures that even when the weighted sum is low or zero, the neuron can still produce a significant output. After calculating the weighted sum of the activations from the previous layer and adding the bias, the neuron passes this value through an activation function f. The activation function is unassuming but is critical as it introduces non-linearity into the neuron's output. Without non-linearity, neural networks could not model the complex, nonlinear relationships often present in real-world data. Activation functions like Sigmoid, Tanh, and ReLU each introduce non-linearity in different ways, affecting the network's learning dynamics and capabilities [1]. Putting this all together, the function for a neuron's activation a can be represented as:

$$a = f(\mathbf{w} \cdot \mathbf{x} + b)$$

In the equation vector, $\mathbf{x}$ represents the neuron's inputs, while $\mathbf{w}$ and b denote the weight vector and bias, respectively [12]. These elements, along with the activation function f (which could be sigmoid, ReLU, or another function), enable the neuron to process and transmit data throughout the network [1]. While a single activation has limited power, the combined action of numerous neurons working together forms the basis of a neural network's remarkable processing power [7].

The interaction of numerous neurons significantly amplifies the computational power of a neural network. Each neuron, with weights $\mathbf{w}$ and bias b, acts as a mini-processing unit [11]. The weights determine how each neuron's inputs are scaled, reflecting the relative importance of each input in the weighted sum calculation. The bias enables the neuron to adjust its output independently of its inputs, ensuring that neurons can activate effectively under various input conditions. Activation functions, such as sigmoid, ReLU, and many more, introduce necessary non-linearity, enabling the network to capture complex and nuanced patterns in the data.

When the data enters the network, it undergoes a series of transformations as each activation is calculated. Every neuron in a layer processes the data through its weighted sum, bias, and activation function, with the output subsequently serving as the input for the next layer [7]. This cascading effect, where a controlled output of one layer becomes the input for the next, allows the network to build an increasingly sophisticated understanding of the input data. Notably, a single change to a weight or bias within any layer can propagate through the network, influencing its overall behavior. This intricate process can be represented as a nested function, where the output of one layer becomes the input for the next, as shown in the following equation:

$$\hat{\mathbf{y}} = f\left(W_3 \cdot f\left(W_2 \cdot f\left(W_1 \cdot \mathbf{x} + \mathbf{b}_1\right) + \mathbf{b}_2\right) + \mathbf{b}_3\right)$$

Here, W_1, W_2, and W_3 represent the weight matrices for successive layers, $\mathbf{b}_1$, $\mathbf{b}_2$, and $\mathbf{b}_3$ are the bias vectors for each layer, and f denotes the activation function applied to each weighted sum plus bias [7]. This layered structure underpins the neural network's ability to handle complex tasks by progressively refining and transforming the input data through each layer's neuron activity. While individual neurons have limited power on their own, the combined action of countless neurons working together in this intricate architecture allows them to learn complex relationships and patterns from the data, leading to the remarkable processing power and capabilities of neural networks.

2.2.2 Weights and Biases

As previously discussed, weights and biases are essential components of neural networks, enabling them to learn complex relationships. Focusing initially on the weights, they are numerical values that represent the connection's strength between two neurons. Higher weights denote stronger connections, resulting in a greater influence on the activation of connected neurons [7]. In a way, the weights act like votes, determining the most and least important inputs. During training, weights are continuously adjusted as new data becomes available, allowing the network to better represent it. This process essentially encodes patterns within the network's weights, leading to an intricate web of connections with varying strengths. This network architecture allows the model to represent and understand complex relationships within the data. The final weight values reflect the network's acquired knowledge, which can be utilized in various applications, like making predictions.

When an input is passed through a network, the weights are used to calculate the weighted sums of connecting neurons. Consider the network in Figure 1 with two input neurons, three hidden layer neurons, and two output layer neurons. To calculate the weighted sums of the second-layer neurons, the column vector containing activations of the first-layer neurons must be multiplied by a weight matrix W containing the six weights w associated with these connections [13]. Given a network with two input neurons and three hidden layer neurons, a 2×3 weight matrix represents the connections between the input and the first hidden layer. This weight matrix can be visualized as:

$$\text{input} \rightarrow \text{hidden} = W = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}$$

In the weight matrix W, the number of rows represents the number of neurons in the input layer, whereas the number of columns represents the number of neurons in the first hidden layer. For example, the weight in the lower right-hand side of the matrix w_{23} connects the second neuron in the input layer to the third neuron in the hidden layer [13]. To calculate the weighted sum for each neuron in the hidden layer, the transpose of the weight matrix $W^\top$ is multiplied by the input vector $\mathbf{x}$ from the input layer.

$$W^\top \cdot \mathbf{x} = \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 \\ w_{21}x_1 + w_{12}x_2 \\ w_{31}x_1 + w_{32}x_2 \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}$$

The resulting vector $\mathbf{z}$ from the operation $W^\top \cdot \mathbf{x}$ represents the weighted sums z for each neuron in the first hidden layer and forms the foundation for the subsequent calculation of their activations [13]. The next step in computing neuron activations involves the integration of biases. Traditionally, every neuron, except those in the input layer, has a bias that is refined during training. These biases are added to the weighted sums from the previous layer's output, thus shifting the activation function. Continuing with the 2-3-2 network example, given that the second layer comprises three neurons, the corresponding bias vector will contain three values, one for each neuron [13]. The biases are integrated with the weighted sums as follows:

$$\mathbf{z} + \mathbf{b} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} z_1 + b_1 \\ z_2 + b_2 \\ z_3 + b_3 \end{bmatrix}$$

In this formulation, $\mathbf{z}$ represents the vector of weighted sums, and $\mathbf{b}$ is the vector of biases. Each element of the resulting vector represents the activation function input for each neuron in the layer. Incorporating biases shifts the weighted sum of inputs from the previous layer, thus establishing a threshold that must be surpassed for significant neuron activation [7]. This mechanism not only ensures that neurons are dynamically responsive to inputs but also gives them the ability to activate meaningfully even in the absence of substantial weighted input. This aspect of biases is crucial for the model's adaptability, enabling it to model complex patterns accurately.

Throughout the training phase, both weights and biases undergo iterative adjustments, primarily guided by the gradient descent algorithm. This optimization technique progressively refines each weight and bias, shaping the influence exerted by neurons on each other. During training, the network strengthens

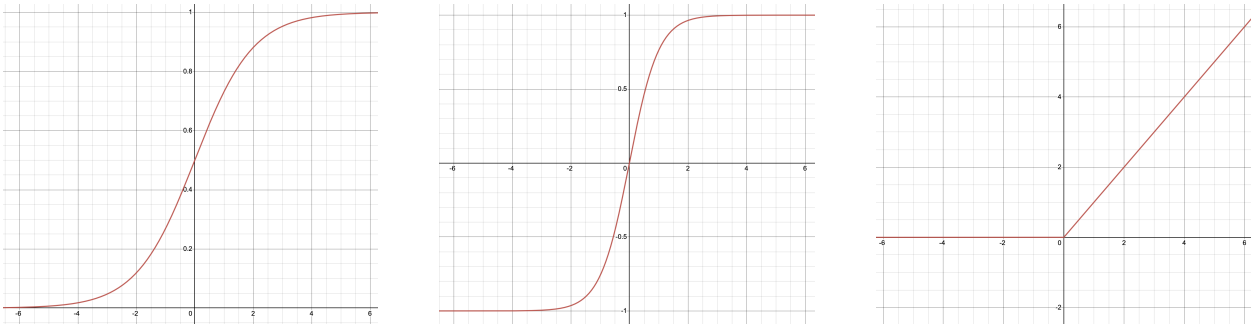
relevant connections and diminishes less significant ones, guiding data transformation from input to output. The nuanced calibration of weights and biases not only affords significant flexibility to neuron activations but also plays a critical role in the network's ability to model intricate data patterns with precision.

2.2.3 Activation Functions

Wrapped around the weighted sum of the input plus the bias is an activation function [7]. Activation functions are mathematical equations that take in one number and output a different number. This output is the activation of the neuron. Without activation functions, all models would be linear, severely limiting their ability to learn complex patterns. Activation functions solve the linearity problem, introducing nonlinearity into the model and allowing them to learn significantly more complex things. In addition to introducing nonlinearity, activation functions play a role in regularizing the network, helping to prevent overfitting [1]. To complete the illustration of the 2-3-2 network, the activation function f is applied to the combined weighted input and bias, ultimately generating the activations of the neurons in the first hidden layer, as represented by the following equation:

$$f(W^\top \cdot \mathbf{x} + \mathbf{b}) = f\left(\begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}\right) = f\left(\begin{bmatrix} w_{11}x_1 + w_{12}x_2 + b_1 \\ w_{21}x_1 + w_{12}x_2 + b_2 \\ w_{31}x_1 + w_{32}x_2 + b_3 \end{bmatrix}\right)$$

When building a network, there are several activation functions f to choose from, each with unique properties. Some of the most common activation functions include the Sigmoid, Tanh, and ReLU functions [1]. These functions transform the output in various ways, influencing how much signal is passed to the subsequent layer. For instance, the Sigmoid function, represented as $\sigma(x) = \frac{1}{1+e^{-x}}$ where x is the input to the neuron (as seen in Figure 2.a), has an S-shaped curve [14]. The Sigmoid function maps any input to an output value between 0 and 1, which is especially useful for tasks like binary classification . However, it has its limitations. One notable drawback is its susceptibility to the vanishing gradient problem [7]. This issue arises because, for extremely high or low input values, the weight updates during training become minuscule, impacting the learning process.



(a) Sigmoid function plot (b) Tanh function plot (c) ReLu function plot

Figure 2: Comparative plots of three common activation functions. (a) Sigmoid function, displaying a characteristic S-shaped curve, smoothly increasing from 0 to 1. (b) Hyperbolic tangent (Tanh) function, similar to the sigmoid but ranging from -1 to 1. (c) Rectified Linear Unit (ReLU), which allows only positive inputs to pass and sets all negative values to zero, shown by the sharp transition at the origin [1].

While resembling the sigmoid, the Tanh function has an output range from -1 to 1. An equation representation of the Tanh activation function is $\tanh(x) = \frac{2}{1+e^{-2x}} - 1$ with the resulting graph seen in Figure 2.b) [1]. The zero-centered nature ensures that negative inputs result in values that lean strongly towards the negative end, and likewise, positive values trend towards the positive end. This characteristic often makes it more convenient than the Sigmoid function, as it helps the network make more pronounced decisions. However, like the Sigmoid, Tanh can also be susceptible to the vanishing gradient problem, albeit to a slightly lesser extent.

Relatively new compared to Sigmoid and Tanh, ReLU has quickly established itself as the go-to activation function for many neural network types. An equation representation of the ReLU activation function is $f(x) = \max(0, x)$ with the resulting graph seen in Figure 2.c) [15]. ReLU is computationally inexpensive, enabling models to train faster. It is inexpensive because its operation is straightforward: if the input is positive, it returns that value; if it is negative or zero, it returns zero. However, this simplicity risks that some neurons might never activate during training, essentially becoming dead and ceasing to adjust. Variants like the Leaky ReLU have emerged to address this shortcoming.

The intricate interplay of a neural network's weights, biases, and activation functions constitutes the foundational framework for its learning and prediction capabilities. Weights determine the strength of connections between neurons, constantly adjusting during training to reflect the emergent patterns in the data. Biases, serving as an additional degree of freedom, shift the activation functions to optimize neuron responses, enhancing the network's adaptability. The activation functions themselves, be it Sigmoid, Tanh, or ReLU, introduce nonlinearity into the network, enabling the capture of complex relationships within the data. Once trained, the weights, biases, and activations work together to process and transform raw inputs into

meaningful outputs, enabling the network to learn from and make predictions on complex datasets. This simplified representation of the neural network article serves as a foundational understanding of network architecture, upon which more intricate architectures can be built.

2.3 Gradient Descent

Prior to training, the weights and biases of a neural network are typically initialized randomly. While certain strategies can be employed to generate more favorable initial values, regardless of the initialization method, the network's initial performance is generally poor [16]. However, through the iterative training process, the network incrementally improves as it discovers the optimal combination of weights and biases that connect neurons across different layers. To identify the optimal parameter values, neural networks typically employ the gradient descent algorithm, iteratively refining the weights and biases until the network's performance reaches its peak.

Gradient descent operates in two phases: a forward pass and a backward pass. During the forward pass, data is propagated through the network, with each neuron's output feeding into the next layer. This cascade of activations ultimately culminates in a final output, which is used to evaluate the network's current performance. Once the forward pass concludes, the backward pass commences. This phase updates the network's parameters with the objective of minimizing the loss function, a quantitative measure of the network's performance [17]. Through repeated cycles known as epochs, the network progressively learns, refining its parameters better to represent the underlying relationships within the training data [7].

2.3.1 Loss Functions

Before updating the weights and biases in a network, it is essential to understand how far the network's predictions deviate from the actual values [7]. This discrepancy is quantified using a loss function that serves as a feedback mechanism for the network. One way to quantify the difference between the actual activation of a neuron and the desired activation is the squared error (SE), which can be calculated as:

$$\text{SE} = \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$$

Where y represents the desired activation level for a neuron, and $\hat{y}$ represents the neuron's actual activation based on the given input. The desired activation for any given neuron will depend on the input. For instance, in classification tasks where each neuron in the output layer is associated with a specific class,

the ideal output for the correct class would be close to 1, with other neurons being close to 0 [7]. The activations in the output layer can be interpreted as the probability of the input belonging to the class the neuron represents. For instance, in classifying handwritten numbers, if the input is an image of the number 8, the activation for the neuron associated with '8' should ideally be 1, and the rest 0. However, when evaluating a model's performance, the entire set of neurons in the output layer is considered; the consideration is not limited to individual neurons. Instead, a holistic measure such as the Mean Squared Error (MSE) is utilized, which aggregates the squared discrepancies across all output neurons and is expressed as [18]:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n} \|\mathbf{y}_i - \hat{\mathbf{y}}_i\|_2^2$$

In this formula, n represents the total number of samples in the dataset or batch being considered. Each $\mathbf{y}_i$ is the desired output vector for the i^th example, and $\hat{\mathbf{y}}_i$ is the network's actual output vector for the same example. The composition of $\mathbf{y}_i$ depends on the specific task; for instance, in a classification problem, it is often all zero with a single one in the place of the correct class. The MSE extends the idea of the Squared Error (SE) by calculating the average of the squared differences between the actual and predicted outputs across all examples in the dataset or batch, thereby offering a comprehensive measure of the model's accuracy. Additionally, during the training process, the loss is typically computed as an average over batches of data rather than individual data points. This approach, executed iteratively across multiple epochs, enhances the efficiency and stability of the training process. For a neural network with three layers, the loss function might be defined as the average squared difference between the predicted output and the actual target across all examples in a batch, guiding the optimization of the network's parameters for improved performance [7]. An MSE loss function for a three-layer network is given by:

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \left\| f_{\theta^{(3)}}^{(3)}\left(f_{\theta_{(2)}}^{(2)}\left(f_{\theta_{(1)}}^{(1)}(\mathbf{x}_i)\right)\right) - \mathbf{y}_i \right\|_2^2$$

In this formulation, the loss function $\mathcal{L}(\theta)$ is designed to evaluate the model's performance using the mean square error across a batch of inputs. Each layer $f_{\theta_{(1)}}^{(1)}$, $f_{\theta_{(2)}}^{(2)}$, and $f_{\theta_{(3)}}^{(3)}$ processes the input $\mathbf{x}_i$ and contributes to the final output. The discrepancy between this output and the target $\mathbf{y}_i$ is squared to determine the error for each example. Averaging these errors across the batch, the loss function $\mathcal{L}(\theta)$ reflects the average error due to the current settings of the network parameters θ.

By averaging these error terms across batches, $\mathcal{L}(\theta)$ understands the average error attributed to the current parameter configuration. During the training process, the goal is to minimize $\mathcal{L}(\theta)$ by iteratively

adjusting the parameters θ to reduce the average error across the training dataset. This approach evaluates the neural network's performance over the entire training data set across multiple training rounds or epochs. In a complete epoch, all batches (each being a subset of the training data) are processed through the neural network. Using this approach, the model parameters are updated after every batch instead of after each piece of training data, which is more efficient, among other benefits. As the average loss decreases, the model's accuracy generally improves across all samples.

2.3.2 Backpropagation

One of the most common ways to determine the weights and biases that minimize the loss function is by using backpropagation. At its core, backpropagation computes the gradient of the loss function with respect to each weight and bias in the network by applying the chain rule [17]. This gradient, represented as a multi-dimensional vector, indicates the direction of the steepest ascent in the parameter space. The negative of this gradient, denoted as $-\nabla f$, gives the direction of the steepest descent in the parameter space [19]. This information acts as a guide, deciding how to adjust each weight and bias to minimize the loss function most effectively, ultimately leading to improved network performance. The negative gradient of any function f can be represented mathematically as:

$$-\nabla f = \begin{bmatrix} -\frac{\partial f}{\partial \theta_1} \\ -\frac{\partial f}{\partial \theta_2} \\ \vdots \\ -\frac{\partial f}{\partial \theta_n} \end{bmatrix}$$

The vector representing $-\nabla f$ is composed of the negative partial derivatives of f with respect to a set of parameters $\theta_{(1)}$, $\theta_{(2)}$, and $\theta_{(n)}$. Each element of the vector indicates the rate of decrease of f with respect to changes in each parameter [17]. By taking the negative gradient of the loss function, which measures the discrepancy between the predicted and desired outputs, we obtain a crucial direction vector. This vector indicates the direction in which each network parameter $\theta_{(i)}$ must be adjusted to minimize the loss function, ultimately leading to a more accurate model [7]. Through an iterative process, the network utilizes the negative gradient information to update its parameters. This process involves calculating the negative gradient at each step and using it to adjust the parameters according to a specific update rule. By iteratively following this process, the network progressively minimizes the loss function and converges towards a local minimum, corresponding to a state of optimal performance. The mathematical expression for updating each parameter $\theta_{(i)}$ is as follows:

$$\theta_i^{\text{new}} = \theta_i^{\text{old}} - \alpha \cdot \frac{\partial f}{\partial \theta_i}$$

In this equation, θ_i^{old} denotes the current value of the i-th parameter in the network, while θ_i^{new} represents its updated value after the execution of a single training step [17]. The term α refers to the learning rate, a hyperparameter that plays a important role in the training process [20]. The learning rate determines the magnitude of the updates made to the network's parameters during each training iteration. A larger learning rate α results in larger steps, enabling the network to traverse the parameter space more rapidly, but potentially overshooting the optimal values. Conversely, a smaller learning rate leads to smaller, more precise steps, reducing the risk of overshooting but potentially prolonging the training process and increasing the risk of getting stuck in local minima.

The expression $\frac{\partial f}{\partial \theta_i}$ represents the gradient of the loss function f with respect to the parameter θ_i [19]. The negative gradient, with respect to parameter θ_i, indicates the direction in which the parameter should be adjusted to achieve the most significant decrease in the loss function f [7]. Moving the parameter θ_i in the direction of this negative gradient will reduce the error between the network's predictions and the actual target values. During training, this update rule is applied iteratively to each parameter in the network. Initially, neural networks often use a relatively high learning rate to allow for rapid exploration of the parameter space. As training progresses and the network converges towards optimal parameter values, the learning rate is typically reduced to allow for more precise adjustments [20]. This strategy balances efficient learning and precise convergence to an optimal set of parameters.

One inherent challenge when utilizing gradient descent with backpropagation is the tendency to converge to local minima rather than the global minimum. Differentiating between local and global minima is not feasible due to the high-dimensional nature of the parameter space in neural networks, where finding and comparing all minima is impossible. As a result, when a new minima is found, there is no way of knowing if it is the global minima or just a better local minima. To increase the likelihood of approaching a global minimum, or at least a more favorable local minimum, advanced variants of gradient descent, such as ADAM (Adaptive Moment Estimation), are employed [21]. ADAM extends the basic gradient descent algorithm by integrating elements of momentum to help the optimization process not get trapped in suboptimal local minima. Consequently, ADAM can navigate the loss function more effectively, increasing the chances of reaching the global minimum or arriving at a local minimum closer to the global optimum regarding the loss function value.

2.3.3 Mathematical Formulation

The mathematical framework underpinning neural networks centers around manipulating weights and biases to adjust neuron activations to minimize the loss function [7]. Despite the inability to directly alter activations, manipulating weights and biases provides an indirect means to influence them and thus affect the network's output. Consider the simple feed-forward neural network illustrated below in Figure 3, which consists of a single input, one hidden layer with one neuron, and an output layer, also with one neuron. Each neuron is connected by weights w_1, w_2, and w_3 and adjusted with biases b_1, b_2, and b_3. An activation function f is applied to each neuron's weighted input z_i to generate its activation a_i.

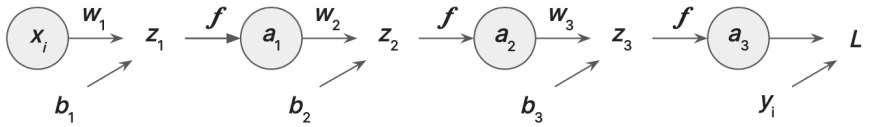


Figure 3: Schematic of a simple feedforward neural network with a 1-1-1-1 architecture. The input x is transformed into the output $\hat{y}$ through a series of weighted sums and activations. Each neuron's output is the result of applying the activation function f to its weighted input z_i, with the final neuron producing the network's output.

In the simple 1-1-1-1 network depicted above in Figure 3, the activations a_1, a_2, and a_3 correspond to the two hidden, and one output layer while x_i is the input layer [17]. The loss for the network is determined by the discrepancy between the desired output y and the output of a_3, which is equal to $\hat{y}$. The discrepancy between y and $\hat{y}$ can be quantified by taking the mean squared error. The process to calculate $\hat{y}$ begins by calculating the weighted sum z_1 for the first hidden layer neuron, which involves multiplying the input activation x_i by the weight w_1 and adding the bias b_1. The resulting z_1 is then passed through the activation function f to produce a_1, the activation of the first hidden layer neuron. This activation serves as the input for the next layer, where it is similarly transformed by weight w_2 and bias b_2 to compute z_2 and subsequently passed through f to obtain the activation of the second hidden layer a_2. This process is then repeated one more time to calculate a_3, which will serve as the final output $\hat{y}$.

Understanding the loss function's construction makes it easy to examine the impact of changing any weight or bias on the overall loss. For instance, to assess the effect of weight w_3 on the loss, the chain rule is applied to decompose the gradient of the loss function with respect to w_3 [19]. This decomposition reveals the pathway through which w_3 influences the loss: from affecting the weighted sum z_3, through the activation a_3, and finally to the loss itself. To minimize the loss, w_3 is adjusted in the direction opposite to its gradient and scaled by a learning rate α [20].

This basic concept also applies to more complex networks, as the principles governing the calculation of gradients remain the same. Consider a three-layer network as an example. The gradient of the loss function with respect to any parameter θ within the network is given by the following general formula:

$$\frac{\partial}{\partial \theta} \left\| f^{(3)}_{\theta_{(3)}} \left(f^{(2)}_{\theta_{(2)}} \left(f^{(1)}_{\theta_{(1)}}(x) \right) \right) - y \right\|_2^2$$

In the equation, $f^{(1)}, f^{(2)}$, and $f^{(3)}$ represent the operations performed within the corresponding network layers, including weighted sum computations, activation function applications, and any other layer-specific transformations [7]. This equation demonstrates how the parameter θ, influences the network's output through each layer's transformation, ultimately impacting the overall loss in relation to the target y. It highlights the interconnected nature of neural network operations and emphasizes how each layer contributes to the final result.

2.4 Forward-Forward

The Forward-Forward (FF) algorithm, introduced by Geoffrey Hinton in 2022, presents a novel approach to training neural networks [4]. It aims to address some of the limitations associated with traditional gradient descent algorithms and potentially provide a closer representation of how the human brain learns. The FF algorithm replaces the traditional forward and backward passes required to perform gradient descent with two forward passes: one using real data (positive data) and the other using fake data (negative data). The FF algorithm aims to train the network to achieve high goodness scores for positive data and low goodness scores for negative data, thereby teaching it to distinguish between the two classes. The dynamic interplay between positive and negative data drives the learning process of the FF algorithm, which can be visualized in Figure 4.

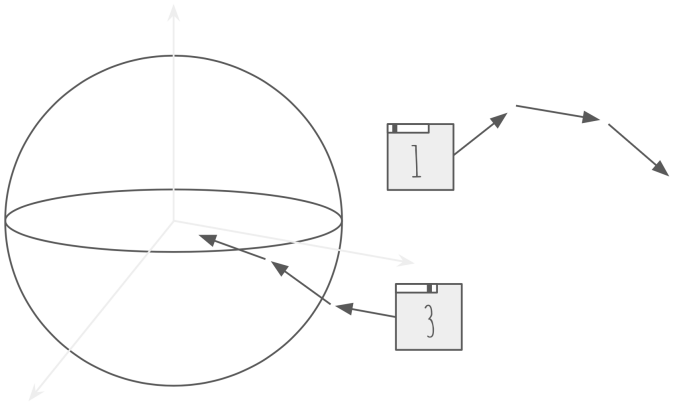


Figure 4: Visualization of neural network training with the Forward-Forward algorithm. During training, positive data '1' moves away from the center, indicating increasing goodness, while negative data '3' moves towards the center, reflecting a decrease in goodness

The figure illustrates the ideal progression of the training process using the Forward-Forward (FF) algorithm with two distinct pieces of data. The depiction of the number 1 is correctly labeled, signifying positive data, whereas the depiction of the number 3 is considered negative data due to incorrect labeling. The proximity of these images to the center of the sphere correlates with their goodness score. The closer an image is to the sphere's center, the lower its goodness, and the more likely the network will classify it as incorrectly labeled. At the outset of training, both pieces of data are positioned equivalently relative to the sphere's center, reflecting the network's initial inability to differentiate between positive and negative data, which yields similar goodness scores. However, as training advances, the network parameters are incrementally updated, as indicated by the directional arrows. Through these updates, the network gradually learns to distinguish between the data types: the representation of the number 3 moves toward the origin, indicating a decrease in its goodness score, while the representation of 1 moves away from the center, indicating an increase in goodness. This dynamic—the divergence in the goodness scores between positive and negative data—is precisely the behavior the FF algorithm aims to achieve. By adjusting the network's parameters to enhance the goodness of positive data and reduce that of negative data, the FF algorithm trains the network to recognize patterns to classify input data as correctly or incorrectly labeled.

With its unique approach to training and the elimination of a backward pass, the Forward-Forward algorithm also removes the need for a global loss function. Instead, the network is trained in a layer-wise fashion, with each layer having its own loss function. As a result, each layer is trained independently, eliminating the need for the network to pause and calculate the overall loss of the network and backpropagate errors. This local learning mechanism potentially aligns more closely with the brain's learning processes compared to the global optimization approach of backpropagation. However, there is a nuanced issue to

consider with this approach. For example, if the network receives positive data and the calculated goodness for a layer is high, this indicates that the layer has accurately identified the data as positive. If all activations are forwarded to the next layer without modification, it becomes very easy for the subsequent layer to recognize that this is positive data, as the prior layer has already optimized all weights and biases to maximize the goodness. A potential solution, as proposed in Hinton's paper, is to normalize the length of the hidden vector before it serves as input to the next layer[4]. Through normalization, all information about the activation magnitude is removed, ensuring that each layer actively participates in the identification process rather than simply relying on the previous layer's output.

2.4.1 Positive and Negative Data

The FF algorithm stands apart from other training algorithms in its input requirements, as it requires both positively and negatively labeled data [4]. Positive data, often referred to as real data, is genuine and correctly labeled. In contrast, negative data, sometimes called fake data, can take various forms and be generated through different methods. The simplest method to create negative data is by intentionally mislabeling positive data. More sophisticated techniques have also been developed. For instance, Geoffrey Hinton explored a method where two images are combined using masking, and a label is randomly applied to the resulting image.

By integrating labels directly into the data, the FF algorithm eliminates the need to compare the network's output with the data labels. This integration enables the network to distinguish between data that is correctly labeled and data that is not. Each unique category y is assigned a distinct label. This label is then combined with the input vector $\mathbf{x}$ to create augmented data. For positive data, the appropriate label is used, whereas negative data is assigned either an incorrect or a random label, depending on the type of negative data being processed. To incorporate the label, positive and negative data vectors $\mathbf{x}$ are combined with their corresponding label vectors $\mathbf{y}$ to form a single vector $\mathbf{z}$, which serves as the input for the network. To compute $\mathbf{z}$, some elements of $\mathbf{x}$ are replaced with elements from $\mathbf{y}$. When represented in matrix form, the combined vectors are structured as seen in Figure 5:

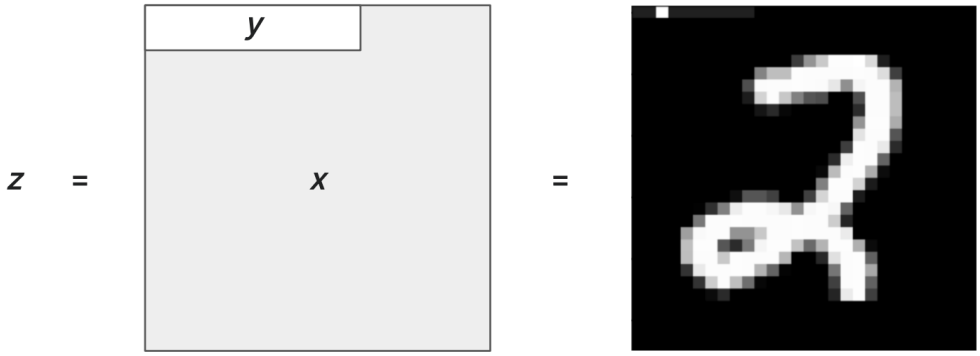


Figure 5: A visual representation of the vector z, which is composed of a data vector x and a label vector y. To the right, an example is provided, showcasing the labeling of a grayscale picture of a handwritten digit two. The correctly labeled greyscale image is an example of images in the set $\bar{\Omega}$.

In this labeling process, two distinct sets of data are created. The first set, denoted as Ω, is defined as the collection of all incorrectly labeled images. The second set, $\bar{\Omega}$, comprises all correctly labeled images. These two sets, with labels embedded directly in them, provide the network with essential information, aiding in learning features associated with each label. As the network trains, it becomes more skilled at distinguishing correctly labeled data from its incorrectly labeled counterparts. The inclusion of negatively labeled images introduces a form of noise that the models must learn to differentiate. Additionally, the negative data helps prevent overfitting and smooths the learning process by forcing the model to consider how changing parameters to benefit positive data identification might adversely affect its ability to identify negative data.

It is important to note that, at this time, a standard method for labeling data has yet to be established. A common approach for different classes is to create a vector where each position corresponds to a different class, indicated by a 1, while the rest are marked with 0 [22]. However, as the number of data classes increases, this method would involve replacing an ever-growing number of pixels in the image with a label, thereby limiting the number of classes that can be effectively represented. To address this issue, using labels that incorporate patterns could be more efficient. This approach would reduce the number of pixels altered by the labeling process, offering a more scalable solution.

2.4.2 Goodness

Goodness is a metric designed to quantify the collective activity of a group of neurons [4]. It reflects the sum of individual neuron activations, with a higher sum indicating greater goodness and vice versa. The idea of goodness is central to the FF algorithm, as its goal is to maximize the goodness of positive

data while minimizing the goodness of negative data. Although the exact goodness value does not directly influence weight and bias adjustments, its gradient provides the necessary information to adjust the network parameters depending on the input. These adjustments aim to optimize the goodness towards an outcome that aligns with the correct classification. In his paper, Hinton recommends calculating the goodness using the sum of squared neural activities, which this project adopts. The goodness for a specific layer is computed as follows:

$$Goodness_{\text{layer}} = \sum_{i=1}^{n} a_i^2$$

Here, a_i represents the activation of the i-th neuron within the layer, and n signifies the total count of neurons in that layer. This formula calculates the sum squared of squared activations, capturing the layer's overall activation intensity of the layer. Once the network is trained, goodness plays a critical role in the prediction phase. While individual layer goodness is informative, the total goodness of the network is typically more relevant for predictions as each layer is trained independently [4]. For a neural network comprising three layers, the aggregate goodness can be computed using the following expression:

$$Goodness_{\text{network}} = \left\| f^{(3)}_{\theta_{(3)}} \left(f^{(2)}_{\theta_{(2)}} \left(f^{(1)}_{\theta_{(1)}} (\mathbf{x}_k) \right) \right) \right\|_2^2 + \left\| f^{(2)}_{\theta_{(2)}} \left(f^{(1)}_{\theta_{(1)}} (\mathbf{x}_k) \right) \right\|_2^2 + \left\| f^{(1)}_{\theta_{(1)}} (\mathbf{x}_k) \right\|_2^2$$

In this equation, each $f^{(i)}_{\theta_{(i)}}$ denotes the operation of the i-th layer, parameterized by $\theta_{(i)}$, where $\mathbf{x}_k$ represents the input vector to the first layer. The equation defines the total goodness score of the network as the sum of the squared L_2 norms of each layer's output, resulting in a holistic measure of the network's total goodness. This total goodness can then be used to make predictions. For example, if all possible labels are applied to an image and a goodness score is calculated for each, the image with the correct label should exhibit the highest goodness, signifying the network's prediction. Furthermore, it is conceivable that the individual layer goodness scores could be manipulated, potentially through scaling or by applying a function, to influence their contribution to the overall network goodness. Such strategic manipulations could be used to emphasize or de-emphasize the influence of particular layers on the model's predictions, offering an increased level of control.

2.4.3 Loss Function

In contrast to traditional training approaches, which employ a single loss function encompassing the entire network, the FF algorithm assigns an individual loss function to each layer [4]. This practice eliminates the need for backpropagation to calculate the gradient of the loss with respect to all parameters, as only one layer is considered at a time. With a loss function placed on each layer, the error does not need to be distributed from the final output back to the first layer, eliminating the need for a backward pass. The FF algorithm operates iteratively, passing the data from the input layer to the first hidden layer. At this point, the goodness of the layer is calculated, the gradient of the goodness function is taken, and corresponding weights and biases are updated to move the goodness of the layer in the desired direction. Once the first hidden layer is trained, the process is repeated for each subsequent layer, treating the previous layer's output as the input for the current one. This layer-wise approach leads to entirely local updates, where each layer focuses on optimizing its own individual objective instead of a global error. For example, a network with three hidden layers would have three separate loss functions, each associated with a different layer. Each loss function computes the goodness of its assigned layer and is used to calculate the gradient with respect to the assigned layer's parameters. The first loss function is specifically designed to optimize the parameters of the first layer:

$$\mathcal{L}^{(1)}(x_k) = \frac{\partial}{\partial \theta_1} \left\| f^{(1)}_{\theta_{(1)}}(x_k) \right\|_2^2$$

This function calculates the gradient of the loss for the first hidden layer by first determining the layer's goodness quantified by squaring the L_2 norm of the layer's output. In this instance, $f^{(1)}_{\theta_{(1)}}(x_k)$ denotes the function representing the output of the first layer when given an input x_k, parameterized by $\theta_{(1)}$. The gradient $\frac{\partial}{\partial \theta_1}$ computes the rate of change of the first layer's loss with respect to its parameters θ_1 [7]. The gradient calculated using this equation can subsequently be used to adjust the first layer's parameters to enhance or diminish the goodness as appropriate. The gradient of the single layers loss function builds upon that of the first using the output of the first layer as the input to the second:

$$\mathcal{L}^{(2)}(x_k) = \frac{\partial}{\partial \theta_2} \left\| f^{(2)}_{\theta_{(2)}} \left(f^{(1)}_{\theta_{(1)}}(x_k) \right) \right\|_2^2$$

The gradient of the second hidden layer $\mathcal{L}^{(2)}(x_k)$ takes the same approach as the first hidden layer but is exclusively used to optimize the parameter of the second hidden layer [4]. Evaluating the goodness of the second hidden layer is slightly more complex, as it also depends on the output of the first layer.

The function $f^{(2)}_{\theta_{(2)}}$ represents the output of the second layer, which is a function of the output of the first layer,$f^{(1)}_{\theta_{(1)}}(x_k)$ for given input x_k. Finally, the gradient of the third layer with respect to its parameters can be calculated with the following equation:

$$\mathcal{L}^{(3)}(x_k) = \frac{\partial}{\partial \theta_1} \left\| f^{(3)}_{\theta_{(3)}} \left(f^{(2)}_{\theta_{(2)}} \left(f^{(1)}_{\theta_{(1)}}(x_k) \right) \right) \right\|_2^2$$

Just like the gradient of loss for the first and second hidden layers, the gradient of the third hidden layer's loss function can be used to optimize its parameters. The third hidden layers loss function incorporates the outputs of the second layer, which incorporates the output of the first layer. This cascading of outputs can be used to add any number of layers. With each layer having its own loss function, a targeted optimization of each layer's parameters is possible. This method simplifies the learning process and ensures the output of each layer is optimized before being used as input for the next. As mentioned, the gradients of the loss functions can be used to update the parameters of each specific layer. In a traditional network, the parameters are adjusted to minimize the loss function. However, depending on whether the data is positive or negative, the FF algorithm will either want to maximize the loss for positive data, equivalent to increasing the goodness, or minimize the loss of negative data, decreasing the goodness. The FF algorithm's unique approach to parameter updates is encapsulated in the equation where Ω are the indices of the incorrectly labeled images and $\bar{\Omega}$ are the indices of correctly labeled images:

$$\theta_i^{\text{new}} = \theta_i^{\text{old}} - \alpha \left(\sum_{k \in \Omega} \frac{\partial \mathcal{L}^{(j)}(x_k)}{\partial \theta_{(i)}} - \sum_{k \in \bar{\Omega}} \frac{\partial \mathcal{L}^{(j)}(x_k)}{\partial \theta_{(i)}} \right)$$

In this equation, the parameters are adjusted based on the difference in the gradients of loss function j for incorrectly and correctly labeled images. θ_i^{old} denotes the current value of the i-th parameter in the network, while θ_i^{new} represents its updated value after the execution of a single training step. The term α refers to the learning rate. The first summation term $\sum_{k \in \Omega} \frac{\partial \mathcal{L}^{(j)}(x_k)}{\partial \theta_{(i)}}$ aggregates the gradients of the loss function $\mathcal{L}^{(j)}(x_k)$ for all data points x_k in the set Ω, representing incorrectly labeled images. In this situation, j specifies the layer for which the gradients are being computed. The second summation $\sum_{k \in \bar{\Omega}} \frac{\partial \mathcal{L}^{(j)}(x_k)}{\partial \theta_{(i)}}$ does the same for the set $\bar{\Omega}$, which consists of correctly labeled images.

By taking the gradient of layer j with respect to both positive (correctly labeled) and negative (incorrectly labeled) data, the function aims to reduce the loss for incorrectly labeled images while increasing the loss of correctly labeled ones. This dual-focused approach is essential because it evaluates the network's effectiveness on both data types. Therefore, the resulting gradient difference can be positive or negative,

reflecting the layer's relative performance on each data type. By taking the difference in gradients, the function ensures that the algorithm concentrates its improvement efforts on areas that are most needed without degrading its success in areas where it is already performing well. If the gradient from incorrectly labeled images is larger, it signals a need for more correction in this area, leading to updates that reduce the loss of these images. Conversely, a larger gradient from correctly labeled images suggests good performance, requiring less adjustment. The goal is to strike a balance where the network loss on incorrectly labeled images decreases while the loss on correctly labeled ones increases. This approach transcends overall loss minimization, enabling targeted improvements in areas where the model is underperforming.

2.4.4 Prediction

Once the network has been trained, it can be utilized for predictions. Unlike traditional neural networks, the networks trained using the FF algorithm do not have an output layer in the conventional sense. Usually, a classification network would have one neuron for every potential category represented by $\mathbf{y}$, and for an input $\mathbf{x}$, whichever neuron has the greatest activation is the prediction [7]. However, the FF algorithm does not train a network to predict the category of the data but rather to determine if the data is positive or negative [4].

If the data is deemed positive, it implies that the label correctly identifies the data. On the other hand, if it's negative, the label does not accurately represent the data. Hence, the network's role is to determine if the label matches the data. To predict the category of an input, all possible labels are applied and processed through the network. The label that generates the highest goodness score is considered the predicted outcome. Typically, predictions are based on the overall goodness of the entire network, but alternative approaches are also possible. Additionally, the absence of a strict one-to-one correlation between the number of neurons in the output layer and the number of categories opens up possibilities for networks with varied structures to be developed.

While the goodness score of a labeled input, in a sense, indicates the likelihood that an input is positive or negative — the higher the goodness, the more likely that the input is correctly labeled — there is no certainty in this prediction. In his paper, Hinton proposes using the following function to formalize the idea of predicting the likelihood that an input is positive:

$$p(\text{positive}) = \sigma\left(\sum_j y_j^2 - \theta\right)$$

In the given equation, the logistic function, σ, is applied to the *goodness*, minus some threshold, θ [4]. This equation indicates that the probability of a positive data point increases as the sum of squared activations increases, and the threshold decreases. If the network's output is strongly positive, indicated by large y_j values, then the data point is more likely to be classified as positive, especially if the threshold is low. To gain a better understanding of the threshold's role, refer to Figure 6, which illustrates this concept:

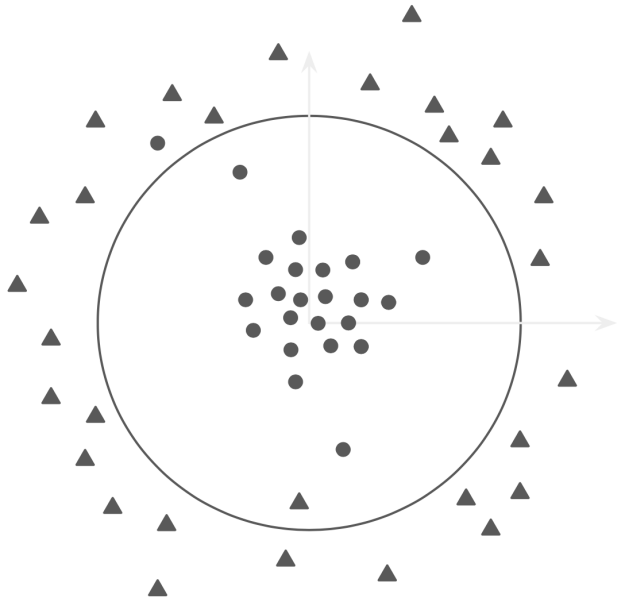


Figure 6: Illustration of the classification of positively and negatively labeled data. Negative data points are represented by circles, while positive data points are depicted as triangles. Points inside the threshold, marked by the large circle, are predicted to be negatively labeled and those outside positively labeled.

In the illustration, negative data is represented by circles and positive data by triangles. Using the method proposed by Hinton, any circle within the larger circle denoting the threshold would be correctly classified (true negative), and any circle outside would be incorrectly labeled as positive data (false positive). Conversely, triangles outside this threshold circle would be correctly classified as positive data (true positive), but triangles inside would be negative data (false negative). If the goodness score equals the threshold, applying the sigmoid σ would yield an output of 0.5, indicating equal probabilities of being positive or negative [1]. When the goodness score exceeds the threshold, the likelihood that the input is positive increases [4]. On the other hand, if this sum is less than the threshold, the probability decreases. Establishing an appropriate threshold is crucial for the network to make accurate predictions. However, determining this optimal threshold is not straightforward. If the threshold is set too high, it may incorrectly claim that positive data is negative, and if it's too low, it may falsely assert that negative data is positive. Depending on the desired outcome, the threshold can be adjusted to alter the distribution of misclassifications.

To determine the optimal value of θ that ensures the network's predictions most closely align with the true labels of the data points, an optimization algorithm such as gradient descent could be utilized. Making predictions using this method would still require applying different labels and comparing the results. However, if the model is sufficiently accurate or the sole objective is to determine whether a single piece of data is positive or negative, not all labels would need to be tested. While this technique has its advantages, mainly due to its simplicity, the first technique was employed for this paper.

2.4.5 Mathematical Formulation

The FF algorithm discards the traditional forward and backward pass mechanism of error propagation in favor of a localized optimization strategy. Here, each layer in the neural network is assigned its own loss function. This approach allows for the independent updating of parameters within each layer, one at a time, thereby negating the requirement for backpropagation [1]. This localized adjustment of parameters simplifies the process, making the influence of changes easier to track. Consider a simple 1-1-1-1 neural network in Figure 7 as an example:

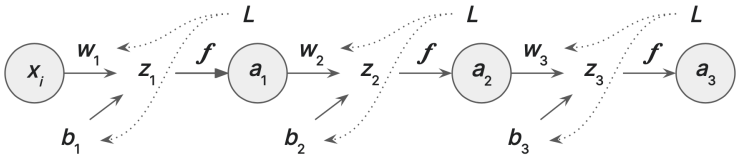


Figure 7: This diagram illustrates a simple 1-1-1-1 architecture neural network used in the Forward-Forward algorithm, showing an input x_i that passes through three successive neurons with activations a_1, a_2, and a_3. Each neuron is associated with its own loss function $\mathcal{L}$, indicating the FF algorithm's unique approach of independent layer-wise optimization without the need for backpropagation.

This illustrative network comprises three neurons with activations a_1, a_2, and a_3. The weights connecting the neurons are denoted as w_1, w_2, and w_3, and biases are represented as b_1, b_2, and b_3. Given that each layer possesses its own loss function, the only parameters influencing the loss are the ones directly connected. For example, the loss on a_2 can only be changed by adjusting b_2 and w_2 because a_1 is locked in place as a result of the layer-wise training process.

The process of updating parameters such as w_2 and b_2 to optimize the loss of the second hidden layer is greatly simplified in this model because it does not require consideration of the entire network's parameters. When calculating the gradient with respect to w_2 and b_2, the chain rule is applied in a localized manner, considering only the immediate inputs and activations, L, z_2, and a_1. This localized gradient calculation means that each layer's parameters are adjusted based on the specific loss function of that layer

alone, thereby streamlining the training process [4]. The gradient indicates the direction of the greatest increase in the loss function, thus allowing the network to adjust the parameters depending on the input. The FF algorithm's layer-specific optimization allows for a more straightforward and efficient training of neural networks.

3 Methodology

3.1 Data

The dataset used in this paper is the widely used MNIST dataset, developed by Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. It is commonly used in the field of machine learning as a starting point for image recognition tasks. It comprises 70,000 grayscale images of handwritten digits ranging from 0 to 9, including those shown in Figure 8. Each image is 28x28 pixels and was originally sourced from a diverse group comprising American Census Bureau employees and high school students. The dataset is pre-divided into a training set of 60,000 examples and a test set of 10,000 examples, providing a structured framework for training and validation [23].

Figure 8: These ten images show a sample from the MNIST dataset representing samples from each of the ten classes. It shows the distinct handwritten digits ranging from 0 to 9, illustrating the variety of styles present in the dataset.

The MNIST dataset was selected for this project due to its straightforward nature, making it an excellent entry point into machine learning, especially for algorithms focusing on image recognition. Its straightforward format allows for easy model building without the complexities inherent in larger datasets. MNIST offers a controlled environment ideal for initial experimentation and algorithm development, making it a natural fit for exploring the FF algorithm.

However, MNIST has drawbacks due to its simplicity, which can lead to models that underperform in complex real-world scenarios. To address the simplicity concern, noise was intentionally added to the MNIST images for a few experiments to present a more challenging and realistic task for the model. These alterations raise the difficulty level, providing a more realistic testing ground for the algorithms and ensuring they are robust and capable of handling the complexities encountered in real-world applications. An example

of data before and after noise is added is shown in Figure 9.

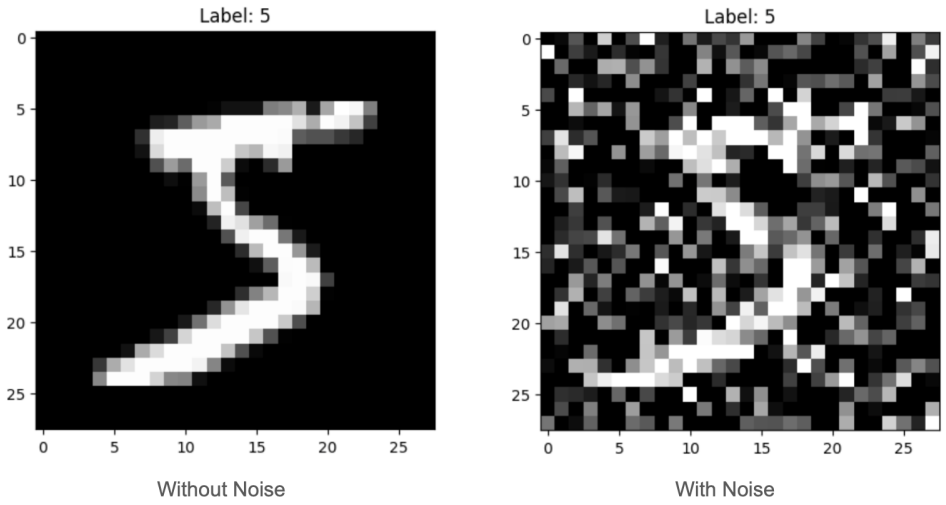
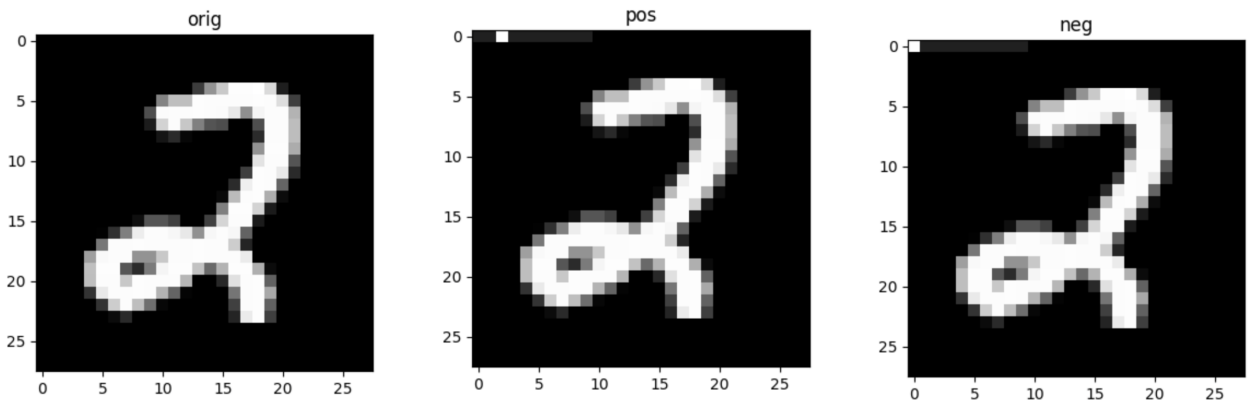


Figure 9: Comparative view of a sample digit from the MNIST in its original form and after applying random noise. The image on the left shows the unaltered image; however, the image on the right shows the same image but with random noise superimposed, more closely resembling real-world data.

3.2 Labeling

To embed the label into the image, the first ten pixels of every image are set to zero [22]. This action creates a blank space where label information can be explicitly encoded. If the image being labeled in a positive manner, the nth pixel corresponding to the image's label is set to the image's maximum value. For instance, if the image represents the number seven, the seventh pixel will be set to the maximum value. The procedure for generating fake data is similar; however, instead of setting the corresponding pixel to its maximum value, a random pixel that doesn't match the correct one is maximized. This process effectively embeds the label into the image in a way that is easily distinguishable by the model during training and inference.

The process is visually demonstrated in Figure 8. Figure 8.a shows the original unlabeled image—a grayscale representation of a handwritten digit two. Figure 8.b illustrates the image with a positive label, where the digit's corresponding pixel in the reserved space is highlighted, clearly indicating the actual label. In Figure 8.c, the negatively labeled image is presented, where a random, incorrect pixel is maximized, thus intentionally misleading the model.



(a) Original unlabeled image. (b) Positively labeled image. (c) Negatively labeled image.

Figure 10: Comparison of original, positively, and negatively labeled handwritten 2 from the MNIST dataset: (a) The original image is unaltered. (b) The positively labeled image is the same as the original, but includes a label with a bright 3rd pixel indicating a 2. (c) The negatively labeled image is identical to the original, but with a label incorrectly indicating that the image represents a digit other than 2.

In the provided example, the digit 2 from the MNIST data set is depicted. The original image is unlabeled and unaltered so it can be converted into either a positively or negatively labeled image. In the positively labeled image, the third pixel is the brightest, indicating that the image represents a 2. It's important to note that the pixel numbering corresponds to the digits 0-9, with the first pixel representing 0. On the other hand, in the negatively labeled or incorrectly labeled image, the first pixel is the most illuminated, incorrectly suggesting that the digit is a 0.

3.3 Procedure

To build and train neural networks, Python was chosen due to its powerful AI libraries and widespread use in machine learning [22]. PyTorch, one of these libraries, is a particularly popular and veritable tool in neural network development. In PyTorch, neural networks are defined as a class inheriting from torch.nn.Module [24]. Within this class, layers such as torch.nn.Linear, along with activation functions like ReLU or Sigmoid, are specified. The class's forward method orchestrates the passage of data through the network's layers. Training within PyTorch includes inputting data into the network, calculating the loss of a loss function, and adjusting the network's parameters accordingly. These adjustments are facilitated by optimization algorithms that PyTorch provides, such as gradient descent or Adam. These algorithms iteratively update parameters to minimize loss, improving the model's prediction capabilities. The implementation of the FF algorithm for this project is based on the PyTorch forward-forward implementation by Mohammad Pezeshki [25].

3.3.1 Dataset Preparation

Before being used as an input to the neural network, the MNIST dataset must be loaded and transformed using a custom MNIST loaders function. The function is defined with two parameters that allow the user to specify the batch size for the training and test data sets. By default, the batch size is set to 50,000 for training and 10,000 for testing.

To begin the transformation process, the images are converted to PyTorch tensors, changing the data type from an image format to a float tensor [24]. Subsequently, the images are normalized so that the pixel values have a mean of 0.1307 and a standard deviation of 0.3081, which are ideal for the MNIST data set [26]. Normalization standardizes the range of the input features, aiding in the convergence of the model and stabilizing the learning process. Finally, the 28x28 pixel images are converted into 784-dimensional column vectors. This flattening process is necessary to align the image data with the input requirements of a fully connected neural network.

In addition to preparing the data set, data loader objects for the training and testing subsets of the MNIST dataset are created. The DataLoader class from PyTorch is used to handle the loading of [24]. The data loader's batch size is set according to the previously defined batch sizes. Shuffling is enabled for the training data loader, which ensures the data is not present in a predictable sequence during training. However, it is disabled for the test data loader as the order of the test data is irrelevant when evaluating model performance. Finally, the MNIST loaders function returns the training and test data loaders ready to be used.

Before discussing the network architecture, the function used to add labels to images must be defined. To create labeled images, a custom data modification function overlay y on x is used. The overlay y on x function takes in two tensors, one containing the data and the other containing the label. The process starts by creating a clone of the tensor containing the image, ensuring the original data remains unaltered. To combine the image and the label, the first ten values of the image tensor are selected and multiplied by 0, effectively zeroing out these pixels. Following this, the function embeds the label into the image tensor. To do this, a specific element in the image tensor is set to the maximum value found in the original image. The element set to the maximum value will correspond with the label tensor. In the end, the function returns the modified tensor containing data from the original tensor with an embed label.

3.3.2 Neural Network Architecture

The neural network consists of two classes: the Net class and the Layer class. It features a straightforward architecture, with an input layer and a user-defined number of hidden layers. The dims argument in the Net class allows the user to specify the number of layers and the number of neurons in each layer. The first layer must always be 784 neurons to accommodate the flattened MNIST dataset images, each being 28x28 pixels.

The layers are instances of the custom Layer class and are sequentially arranged in the Net class, ensuring a streamlined data flow through the network. This linear stacking of layers forms a feedforward neural network architecture, where each layer's output serves as the input to the subsequent layer. Each Layer instance is a specialized subclass of PyTorch's nn.Linear, which performs a linear transformation to the incoming data [24]. Additionally, this class incorporates a ReLU activation function. The ReLU function introduces non-linearity to the model, enabling it to learn more complex patterns in the data. This is important for effectively processing and recognizing the varied patterns present in handwritten digits. Furthermore, the Layer class is uniquely designed with an additional training method. This method, which uses the Adam optimizer, is made to process both positively labeled and negatively labeled samples differently. It optimizes the layer's weights and biases to enhance the divide between correct and incorrect labels.

The Net class also includes a predict method, which differs from traditional prediction mechanisms. Instead of a simple forward pass, this method overlays each potential digit label onto the input data and evaluates the goodness of each label. The label that yields the highest goodness score is selected as the prediction. This approach aligns with the overall design of the network, where the integration of label information directly into the data is a core idea.

3.3.3 Training

Before training the network, two sets of samples from the training data set are prepared. The positive samples are created by applying the overlay y on x function to overlay the correct digit label onto each image. In contrast, the negative samples are generated by overlaying randomly shuffled, incorrect labels onto the images.

The training process begins with preparing a negative and positive set of samples from the training dataset. Positive samples are created by overlaying the correct digit label onto each image using the overlay y on x function. Conversely, negative samples are generated by applying incorrect labels. The actual training

is conducted in a layer-wise manner, a departure from the more common end-to-end training of all layers simultaneously. Each layer is trained independently using a custom method defined in the Layer class. During training, the layer processes both positive and negative samples and calculates a loss function designed to maximize the layer's response to positive samples and minimize its response to negative samples. This loss function is based on the square of the layer's activations.

The training process involves iterating through multiple epochs, during which the layer's weights are adjusted to optimize this loss. The optimization is performed using the Adam optimizer, a popular choice for training neural networks due to its efficiency and adaptive learning rate properties. After training one layer, the process moves to the next, ensuring that each layer learns to extract features effectively before passing its output to the subsequent layer. After training, the model's performance is evaluated on both the training and test datasets, providing insight into the model's generalization capability. For transparency and reproducibility, the implementation details of this project are publicly available. The code, hosted on GitHub, can be found at the following repository: GitHub Repository for PyTorch Forward-Forward MNIST Project. This repository includes all relevant scripts and documentation to replicate and the training process and model evaluation.

3.4 Evaluation

Unlike backpropagation, the FF algorithm trains the network one layer at a time and employs a unique prediction method, necessitating a different approach to tracking accuracy during training [4]. To calculate the network's accuracy, the number of correct predictions is divided by the total number of predictions made. Once the network is fully trained, these predictions are made using all layers combined goodness scores. However, in my implementation, where one layer is fully trained before proceeding to the next, including all the untrained layers in the accuracy calculation doesn't make sense. Instead, accuracy is determined by considering only the trained and currently training layers in the goodness calculations. This method also conveniently illustrates the benefits, if any, of adding additional layers to the network.

4 Results

4.1 Initial Exploration

The exploration of the Forward-Forward (FF) algorithm began with an experiment involving a basic neural network. The main goal of this preliminary network was to assess the feasibility and effectiveness of

the FF algorithm. This goal was achieved by training a straightforward network utilizing the FF algorithm instead of backpropagation. The network consisted of four separate layers: an input layer followed by three hidden layers, each containing 500 neurons. The training process was segmented, with each layer trained for 1000 epochs before moving on to the next one. In order to understand how the network is learning, the accuracy was tracked throughout the training process. The outcomes of this experiment are displayed in Figure 9.

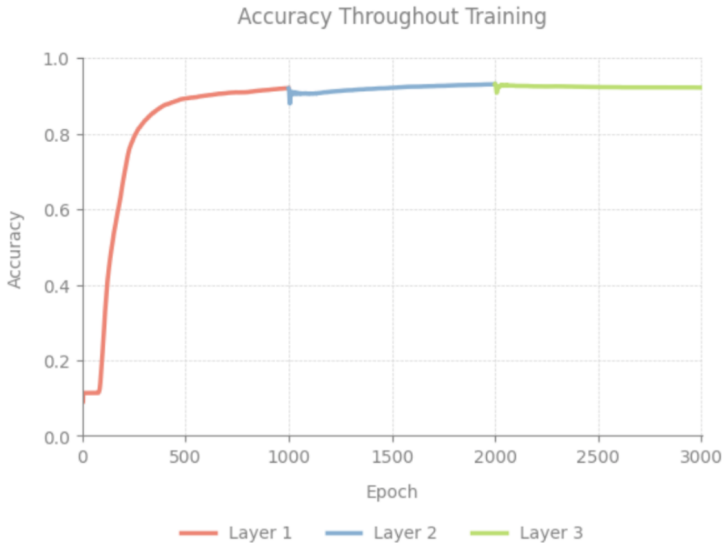


Figure 11: Chart showing the training accuracy of a neural network trained using the FF algorithm across three layers of 500 neurons. The accuracy quickly peaks during the initial training phase and remains consistently high across subsequent layers through 3000 epochs.

As shown, the accuracy changes as training progresses, from the first hidden layer (red), the second hidden layer (blue), and finally on to the third hidden layer (green). The first hidden layer exhibits a rapid increase in accuracy, reaching a plateau near 90 percent shortly after training began. The rapid rise in accuracy suggests that the first layer was able to quickly discern patterns or features in the data that were highly predictive of the desired output. When the second layer was added, the accuracy of the network initially dropped. The initial drop makes sense because new untrained neurons are now used in the goodness calculation to make predictions. Yet, with some training, the accuracy of the network recovers; however, the rate of improvement is more gradual, indicating a different learning dynamic. The final layer shows a pattern similar to the previous layer; however, the initial drop in accuracy is less pronounced, likely due to the reduction in the ratio between the number of neurons added relative to the number of existing neurons. Adding the third layer does little to improve the accuracy of the network.

The training and testing performance of the network offers additional insight into model perfor-

mance. The model achieved a training accuracy of 92.06 percent and a testing accuracy of 92.13 percent. These training and test accuracies indicate that the network was able to learn using the FF algorithm, optimizing its weights and biases to find the ideal loss. The difference of 0.07 percent between the training and testing accuracy shows that the model could effectively generalize to new, unseen data. While the errors are relatively low, they still represent the portion of the data for which the model's predictions were incorrect. Understanding and analyzing these errors could provide opportunities for further refining the model.

4.1.1 Separating Positive and Negative Data

To better understand how the FF algorithm works, it is important to consider more than just the accuracy metrics. A more nuanced insight into model behavior can be gleaned from examining the distribution of goodness scores assigned by the model to positive and negative data. The network is designed to classify images into digit categories (0-9) by testing all possible labels. After processing an image, the network yields ten goodness scores corresponding to the ten-digit labels. The label associated with the highest score is considered the model's prediction. In this analysis, these goodness scores are split into two distinct categories: the 'True Label Scores,' which correspond to the actual, correct label of each image, and the 'False Label Scores,' which are the scores associated with the nine incorrect labels for each image. In an ideal scenario, one would expect to observe higher goodness scores in the 'True Label Scores' category, indicating a high level of confidence that the image is labeled correctly, and comparatively lower scores in the 'False Label Scores' category, implying a lack of confidence that the image is labeled correctly. This disparity in goodness scores is crucial as it underpins the model's ability to categorize images accurately.

To visualize the impact of training on the distribution of goodness scores across correctly and incorrectly labeled images, two histograms were generated before and after model training. The same model described previously was employed for this analysis. A clear separation between the goodness scores of the two categories would indicate the network is accurate and consistently confident in its predictions. Conversely, a significant overlap in the goodness scores across the two categories would suggest the model is having a hard time classifying images, potentially leading to misclassification of specific images.

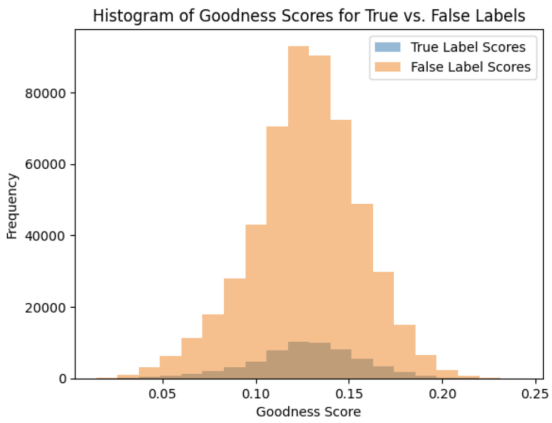


Figure 12: Pre-training histogram of neural network goodness scores, showing an undifferentiated distribution between true (blue) and false (orange) label scores, indicating no initial bias.

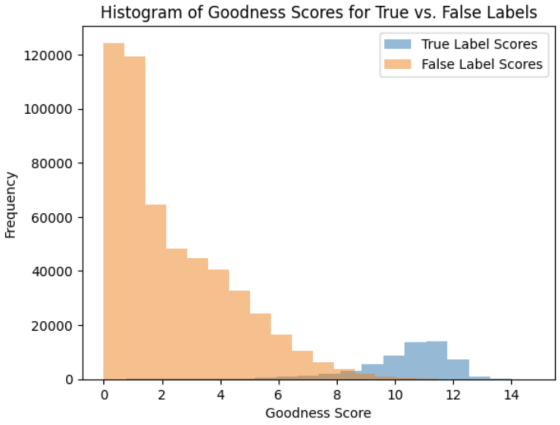


Figure 13: Post-training histogram of networks goodness scores, depicting a clear separation between true (blue) and false (orange) label scores, demonstrating the network's ability to learn.

The histograms in Figure 10 and Figure 11 illustrate the distribution of 'goodness scores' for the neural network's predictions before and after training, respectively. The discrepancy between the number of false label scores and true label scores is due to the fact that 9 out of 10 labels tested for every image are false. Figure 10 displays the initial state of the network, where the goodness scores for both true and false labels follow the same distribution without any discernible separation. This suggests that before training, the model does not show a strong preference for the correct digit class over the incorrect ones, a typical behavior for an untrained neural network that has yet to learn to differentiate between positive and negative data.

After training, as shown in Figure 11, there is a difference in the distribution of goodness scores. The scores for true labels are now separated and generally exhibit significantly higher goodness values, in contrast to the false label scores, which are mostly concentrated near zero. This separation indicates that the training process has enabled the model to assign higher goodness to positive data and lower goodness scores to negative data, demonstrating successful learning. However, the separation between the two classes is incomplete, with some overlap still present. This overlap represents the images that were misclassified by the network. Interestingly, the overlap appears to have a normal distribution, suggesting a similar amount of false positives and negatives.

4.1.2 Average Goodness Scores

To gain a better understanding of the model's misclassifications, an analysis of average goodness scores was conducted. To investigate the model's misclassification, a 10×10 table of average goodness scores

was created, offering an interesting perspective on the model's performance. The table is organized with rows and columns representing digits from 0 to 9. The rows in this table correspond to the true labels of the images. For example, the first row is dedicated to images of the digit 0, the second row for 1, and so on, up to 9. Columns represent how the network interprets these images. Each column shows the average goodness scores assigned by the network when it predicts the images as a certain digit, starting from 0 in the first column to 9 in the last.

Put another way, each cell in this table contains the average goodness score for a specific scenario. The cell at the intersection of row i and column j shows the average score for images that are actually of digit i but are predicted by the network as digit j. The diagonal cells of the table (where $i = j$) are particularly important. High scores along the diagonal indicate accurate and confident predictions by the network. In contrast, high scores in off-diagonal cells (where $i \neq j$) suggest areas where the network may incorrectly assign a high goodness score to the wrong digit. The table as a whole offers a nuanced view of the model's classification behavior across different all numbers, highlighting both its strengths and the specific areas where the model may be prone to errors.

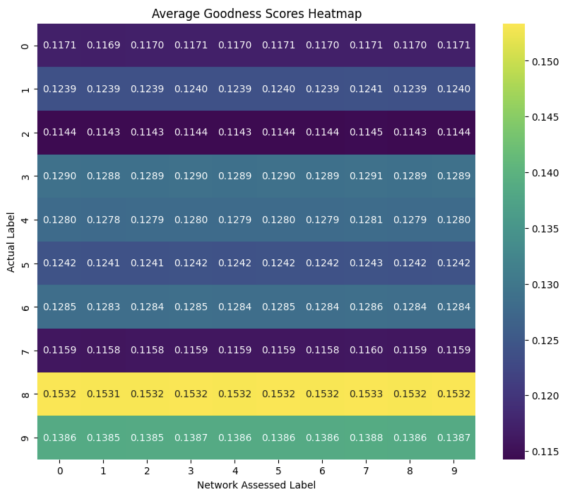


Figure 14: Heatmap of Average Goodness Scores Before Training: This heatmap illustrates the initial state of the neural network, with each cell representing the average goodness score for the corresponding true label (rows) against the predicted label (columns) before any training.

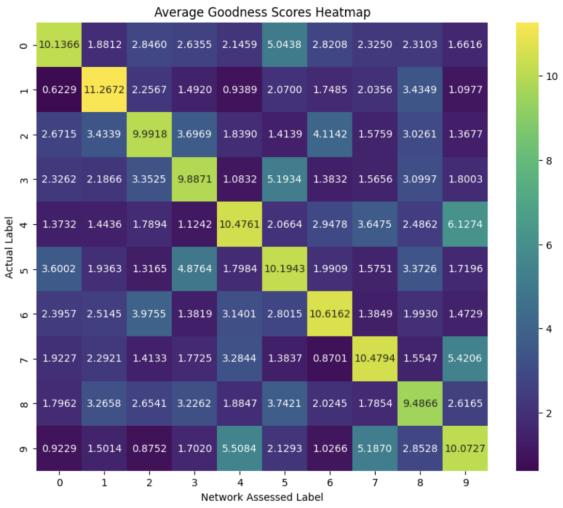


Figure 15: Heatmap of Average Goodness Scores After Training - Post-training, this heatmap displays the neural network's learned ability to correctly classify digits, as evidenced by the prominent diagonal where the true labels match the predicted labels.

Figure 13 and Figure 14 depict heat maps that showcase the evolution of average goodness scores before and after training a neural network. These heatmaps were generated from the same FF-trained network discussed previously, which featured three hidden layers with 500 neurons each and was trained over

1000 epochs, as depicted in Figure 9. The heatmaps visually convey the average goodness scores across the ten possible digit categories to which an image may be classified. In these heatmaps, higher goodness scores are indicated in yellow, while lower scores are represented in purple.

Figure 13, which presents the heatmap prior to training, shows a uniform distribution of low goodness scores across all labels. This uniformity signifies that the network has not yet established the ability to differentiate between the digit classes. The scores are consistently similar and do not vary between the actual and the predicted labels, underscoring the network's initial state of uncertainty in making predictions. This lack of discriminative ability is typical at the beginning of the training process. The uniform lines of color in the heatmap make it look like there is a stark difference between classes, but a closer comparison of the values between them reveals very little difference. This initial state sets the stage for the network to learn and distinguish between the digits as training progresses.

After the training phase, the heatmap, as shown in Figure 14, reveals significant changes in the average goodness scores. The heatmap now features a distinct diagonal element with higher scores for correctly labeled images. This pattern suggests that the network has learned to associate the correct labels with the respective images. In contrast, the off-diagonal elements correspond to incorrectly labeled images and show relatively lower scores. The lower score suggests the network recognizes these as likely being mislabeled, resulting in the lower goodness score. Notably, there is a relative consistency in the average goodness scores for correctly labeled images. However, the sum of averages does not remain constant across rows and columns, suggesting a level of independence in goodness scores. For instance, the digit '9' may receive relatively high goodness scores when misclassified as 4 or 7, yet it maintains a comparable score to other diagonal elements for its correct label, 9.

While the heatmap isn't symmetrical, the trends it reveals are generally bidirectional. Mistakes occur in both directions; for example, the average goodness for a 4 labeled as 9 is 6.1274, and for a 9 labeled as 4, it's 5.5084. Even though these scores are lower than the average score of 10.4761 for a 4 correctly labeled as 4, or the average score of 10.0727 for a 9 labeled as 9, they are substantially higher than the average goodness of an incorrectly labeled image. The smaller gap between these goodness scores suggests that these are areas where the network is more likely to make incorrect classifications, leading to a reduction in accuracy. Aside from mixing up 4s and 9s, the network tends to confuse 7s and 9s, 5s and 3s, and 0s with 5s, but not as much the other way around. This nuanced performance demonstrates the network's learning process, emphasizing the importance of a significant differential in goodness scores for accurate classification.

4.1.3 Goodness by Label

With a clear understanding of the network's strengths and weaknesses, I decided to examine some specific examples. Out of the 10,000 test images, four images representing the extremes of classification were examined. The analysis included evaluating the correctly classified images with the highest and lowest goodness scores, as well as the incorrectly labeled images with the highest and lowest goodness scores. These four images are presented in Figure 16 below.

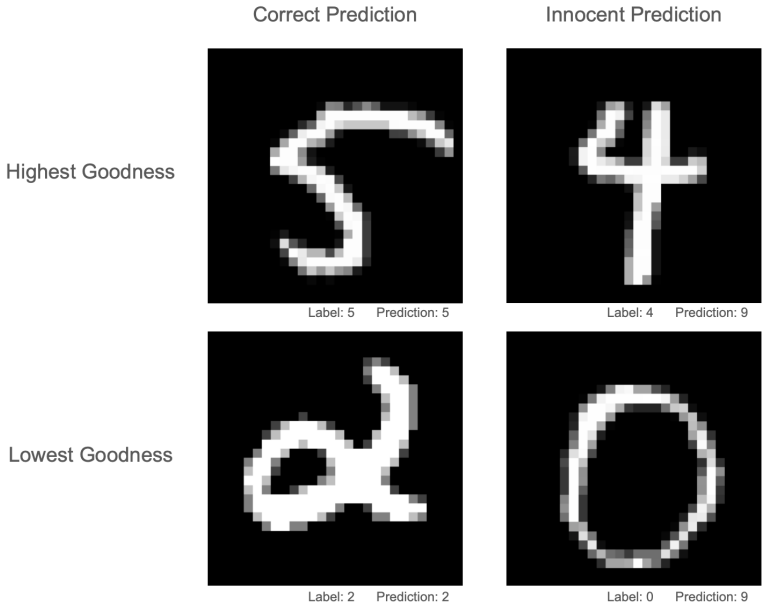


Figure 16: Four images representing the range of the network's classification capabilities. These images were specifically chosen to illustrate the network's performance at its extremes, including the most and least confident correct classifications and the most and least confident incorrect classifications. The images are arranged starting from the upper left, displaying numerals 5, 4, 0, and 2 in a clockwise direction.

Upon reviewing these images, the number depicted in each image becomes apparent. Starting in the upper left with a depiction of the number 5 and moving clockwise, the following images are a 4, 0, and finally, 2. The network accurately identified the 5 and the 2 but incorrectly classified the 4 and the 0. An examination of the distribution of goodness scores for each label, as detailed in Table 2, offers insight into the network's decision-making process.

Label	0	1	2	3	4	5	6	7	8	9
5	6.4488	2.1765	1.8676	8.3286	3.1547	**14.9997**	3.8466	1.4007	4.1651	2.6260
2	0.8783	0.5372	**3.5169**	0.8521	2.8531	1.0885	2.1518	0.4595	1.7455	0.4776
4	5.2864	1.2599	4.6193	2.6863	**5.6702**	1.8669	1.6748	6.1361	1.8740	12.2657
0	**3.4936**	0.1716	0.2495	0.7193	0.4750	2.4908	0.2279	2.5548	0.4202	3.6398

Table 2: Goodness scores assigned by the network for different labels. Each row corresponds to a specific image, while the columns represent the goodness of each label. The highlighted cells (in light gray) show the highest goodness score for each row, indicating the network's final classification for that particular image, and the bold shows the correct label.

The table shows the goodness scores of the four images depicted in Figure 16. Each row of the table corresponds to a specific image, while the columns display the goodness score for each image when a particular label is applied. Recall that all possible labels are tested, and the label with the highest goodness score is considered the network's prediction. The cells highlighted in light grey indicate the network's predictions, whereas the bold scores represent the goodness score of the correct label. For the image representing the number 5, the network correctly identified it as a 5, assigning it the highest confidence score of all the images at 15.00. This indicates a strong certainty in this prediction. In contrast, the 2 image, while also correctly identified, received the lowest confidence score of just 3.52 among all correctly predicted images. This lower score suggests less certainty in this accurate classification, which is understandable when looking at the image.

Continuing counterclockwise, the image meant to depict a 0 was mistakenly classified as a 9. This error occurred with the lowest confidence score among all incorrect predictions, potentially reflecting the network's uncertainty about this classification. The confidence score for the incorrect label 9 was 3.64, compared to 3.49 for the correct label 0. Hopefully, with additional training, this gap will reverse, leading to a correct prediction. Lastly, the image of a 4 was incorrectly identified as a 9, with the highest confidence score among the incorrectly classified images, at 12.26 compared to the correct label's score of 5.67. This outcome is particularly concerning as it signifies a significant misjudgment by the network despite not appearing to be a particularly challenging classification. However, the misclassification is not entirely surprising, as Figure 15 shows that the label 9 on an image of 4 has the highest confidence score for any incorrect label.

4.2 Single Layer Networks

The initial results showed that single-layer networks achieved surprisingly good accuracy before adding additional layers. To further explore single-layer networks, a series of experiments were conducted. A single-layer network with a single hidden layer of 784 neurons, mirroring the size of the input layer, was used as a starting point. This approach, matching the number of neurons in the hidden layer to that of the input layer, aimed to understand the capabilities and limitations of a single-layer network in its simplest form. The network was trained over 5000 epochs, resulting in Figure 17, which depicts the accuracy of the network across these epochs.

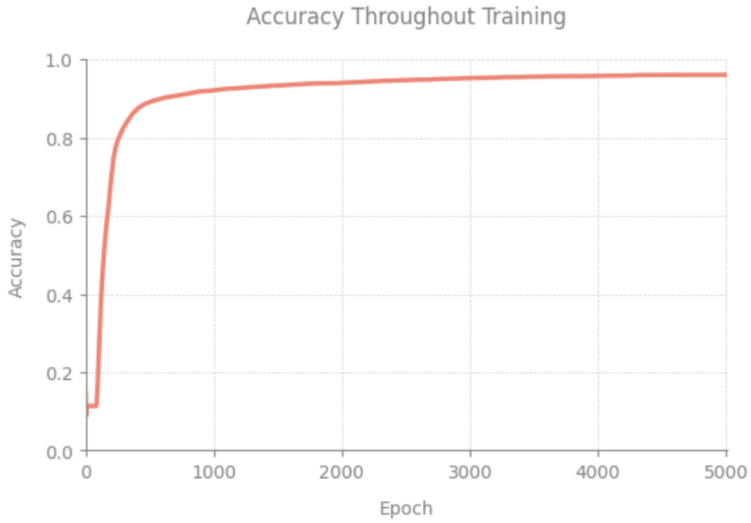


Figure 17: This graph illustrates the accuracy of a single-layer neural network containing 784 neurons trained over 5000 epochs. It demonstrates the network's learning progression, highlighting a steep increase in accuracy during the initial phase, surpassing 90 percent accuracy within just a few hundred epochs.

The single-layer network demonstrated a steep learning curve, with the accuracy rapidly surpassing 90 percent within the first few hundred epochs. This rapid convergence to high accuracy suggests that the single-layer network could quickly capture the underlying patterns in the training data. Following this initial surge, the accuracy continued to improve, albeit at a significantly reduced rate.

Upon completion of training, the network achieved a training accuracy of 97.44 percent. This high level of accuracy during the training phase suggests that the network's simplistic architecture was surprisingly effective at discerning patterns and relationships within the training dataset. When evaluated against the test dataset, the network attained a test accuracy of 96.02 percent, only marginally lower than the training accuracy. This modest discrepancy of approximately 1.42 percent between training and testing performance indicates the model's robustness and ability to generalize well to unseen data.

The performance of the single-layer network suggests that, even without the complexity of additional layers, it can effectively model the relationship between input features and target outputs. The sustained high accuracy beyond the plateau point also implies that the network was neither overfitting nor underfitting the data. However, when noise was introduced to the data, its performance deteriorated: the test accuracy dropped to 91.90 percent, and the training accuracy fell to 82.46 percent. This significant difference in performance indicates that there might be better choices for handling more complex data than a single-layer network. However, the large gap between training and test accuracy indicates overfitting, which might partially explain the drop in preference.

4.2.1 Neuron Quantity

After realizing that a single layer could achieve surprisingly high accuracy, I decided to investigate the effect of neuron quantity on performance. The objective was to determine the impact of varying neuron counts on peak performance and the number of epochs needed to reach optimal performance. For this purpose, several single-layer networks were trained over 5000 epochs, with neuron quantities ranging from 10 to 3000. The accuracy of these networks was monitored throughout the training period, as shown in Figure 18.

Figure 18: The graph compares the evolution of network accuracy over a training period of 5000 epochs for five networks. Each network has a single hidden layer with varying neuron counts, ranging from 10 to 3000 neurons.

All networks tend to converge to similar accuracy levels after a comparable number of epochs, but

those with more neurons generally achieved slightly better accuracy. The 10 and 50 neuron layers performed slightly worse than the larger layers, which interestingly performed nearly identically after 5000 epochs. The accuracies achieved by these networks are displayed in Table 3 below:

Neuron Count	**Training Accuracy (Percent)**	**Test Accuracy (Percent)**
10	90.88	90.24
50	95.29	93.77
200	97.12	95.57
784	97.47	96.08
3000	97.56	95.91

Table 3: A breakdown of the training and test accuracies achieved by a single hidden layer recorded after training for 5000 epochs. It presents data for networks with 10, 50, 200, 784, and 3000 neurons, showcasing the impact of neuron quantity on network performance.

Despite the 10-neuron network having the lowest accuracy among the different configurations tested, it demonstrates that even with a relatively small number of neurons, the network was capable of learning. The 50 and 200-neuron networks showed improved performances, with the best performance coming from the 784-neuron network. Interestingly, the 3000-neuron network performed worse, suggesting that beyond a certain point, increasing the number of neurons does not necessarily translate to better generalization on the test data, which could be a sign of diminishing returns after a certain point when adding neurons. While more neurons in a single-layer network can be beneficial, the improvements in accuracy tend to plateau around 200 neurons, highlighting a potential trade-off between layer size, computational efficiency, and generalization performance.

The performance of the 10-neuron network was notably competitive compared to the larger layers, prompting an investigation into the smallest viable network size before a significant decline in performance is observed. To explore this, ten networks were trained, each with a single hidden layer but varying in neuron count from one to ten. Each network was trained over 5000 epochs. The results of this experiment are displayed in Figure 19 below.

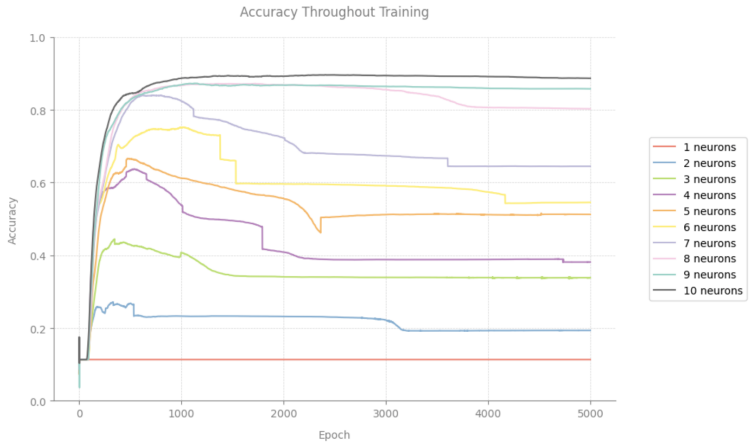


Figure 19: Comparison of neural networks with different neuron counts in the hidden layer, ranging from 1 to 10 neurons. The chart tracks test accuracy for each configuration throughout the training period of 5000 epochs. It visually shows how networks with more neurons tend to achieve higher accuracy, demonstrating the impact of neuron quantity on the network's learning capability.

The graph depicts the test accuracy of the ten networks, with the number of neurons in the hidden layer ranging from one to ten throughout the training period. Initially, all networks experience a rapid increase in accuracy, a trend common across all configurations. However, as training progresses, several networks peak in accuracy and then begin to show a decline, indicative of overfitting. Mid-sized networks seem prone to this pattern, while larger networks are unaffected. The struggle of the midsized networks could be due to their limited complexity, which is insufficient for capturing the necessary level of abstraction in the data, leading them to learn noise instead. Yet, these networks are not so simplistic as to lack the ability to learn. As for the model's accuracy, a clear divergence in performance emerges as the training progresses: networks with a higher count of neurons tend to consistently reach significantly greater levels of accuracy in an order that aligns with their neuron count. Additionally, networks with more neurons learn faster, as evidenced by the steeper learning curves. The train and test accuracies for each network setup are detailed in Table 4 below.

Neurons	1	2	3	4	5	6	7	8	9	10
Train Accuracy	11.254	19.36	33.82	38.296	51.53	54.624	64.726	80.518	86.77	89.538
Test Accuracy	11.35	19.35	33.85	38.15	51.26	54.55	64.47	80.28	85.77	88.67

Table 4: Comparison of training and testing accuracies for neural networks with varying neuron counts in their hidden layer, from 1 to 10 neurons. The table reports the performance of each network configuration after training for 5000 epochs, showcasing a trend of increasing accuracy with higher neuron counts.

A closer look at the training and test accuracy confirms what the chart indicated: networks with

more neurons tend to have higher accuracies. The progression from 1 to 10 neurons displays a consistent increase in performance. Starting with the network containing only one neuron in its hidden layer, it shows an 11 percent accuracy on training and test sets, marginally above the accuracy expected by chance in a 10-class classification problem.

As the neuron count increases, the network's capacity to capture and represent more complex patterns and relationships within the data also grows. This is evidenced by the marked performance improvement observed between networks with a very low neuron count (1-3 neurons) and those with a moderately higher count (4-7 neurons). The rise in accuracy is more notable at the lower end of neuron counts, suggesting that at the beginning, each additional neuron significantly contributes to the network's learning capacity. However, as the neuron count continues to rise, the increments of improvement become smaller, indicating diminishing returns. For example, the performance difference between networks with 9 and 10 neurons is less than the difference between those with 1 and 2 neurons.

The correlation between the number of neurons and the level of performance is explained by the fact that a higher number of neurons enables the network to construct a more complex model of the data. This complexity improves the network's ability to distinguish between the dataset's classes, thus leading to higher accuracy. However, more neurons also mean more parameters to learn, which may require more data and longer training times to be effectively utilized. In this case, the observed increase in performance with neuron count suggests that the training data and epochs are sufficient for the larger models to reach convergence easily.

The trend of diminishing returns as the neuron count increases implies an optimal number of neurons for this specific task and dataset, beyond which additional increases may not result in significant performance gains. Looking back at Table 3, it seems reasonable to infer that the optimal number of neurons is between 200 and 3000, as the configuration with 784 neurons achieves higher test accuracy than any other neuron count examined. This observation is crucial for designing neural network architectures, as it indicates that indiscriminately increasing network size is not always the most efficient strategy. It highlights the importance of finding a balance between model complexity and performance. While these findings are promising, it is essential to remember that the MNIST dataset is relatively simple. Therefore, considering more complex and realistic datasets is important. To address this concern, the same experiment was conducted a second time using noisy data, leading to Figure 20, which compares the training accuracies.

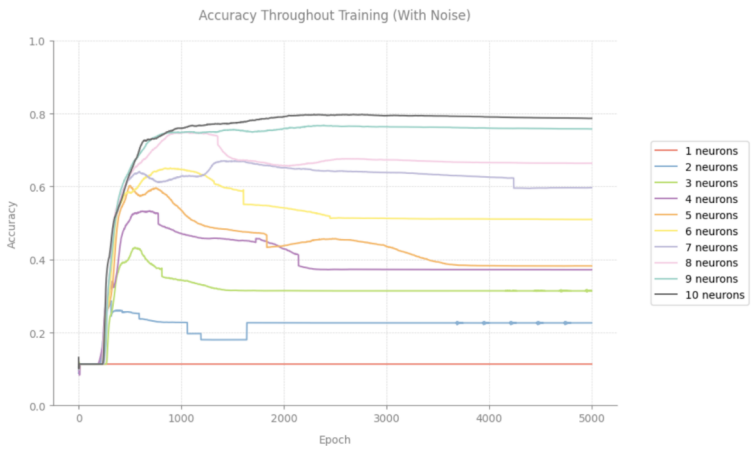


Figure 20: A comparative analysis of neural network performances with noise added to the data employs the same neuron count configurations as in Figure 19. This chart highlights the changes in test accuracy caused by increased data complexity, showing a general decline in performance across all configurations.

Comparing Figure 20, which applies the same methodology but uses noisy data, to Figure 19, which uses clean data, there is a noticeable decline in accuracy. Interestingly, networks with fewer neurons in the hidden layer saw less impact, confirming that they were not learning much. They simply benefit from the default 10 percent chance of picking the correct label in a 10-class classification problem. The most significant loss in accuracy occurred at the higher end, where the 10-neuron hidden layer's accuracy fell by almost exactly 10 percent, resulting in a test error of 87.85 percent. Despite this, the general trends persisted, leading to a squished version of Figure 19. In addition to the drop in accuracy, the learning rate significantly slowed, requiring more epochs to reach maximum accuracy. While a decrease in both accuracy and learning rate is to be expected when the only change is the increased complexity of the data, it highlights the limitations of what a network with a single layer is capable of. Therefore, it is worth exploring multi-layer networks.

4.3 Multi Layer Networks

While a lot can be achieved with single-layer networks, they have limitations as data becomes more complex. With additional layers, the network should be able to model more complex representations of the data. To begin exploring how multi-layer networks could improve accuracy when encountering more challenging data, a simple network with five hidden layers, each containing 500 neurons, was created. Each network layer was trained for 500 epochs before proceeding to the next, using noisy data. The resulting network's accuracy throughout training is depicted below in Figure 21.

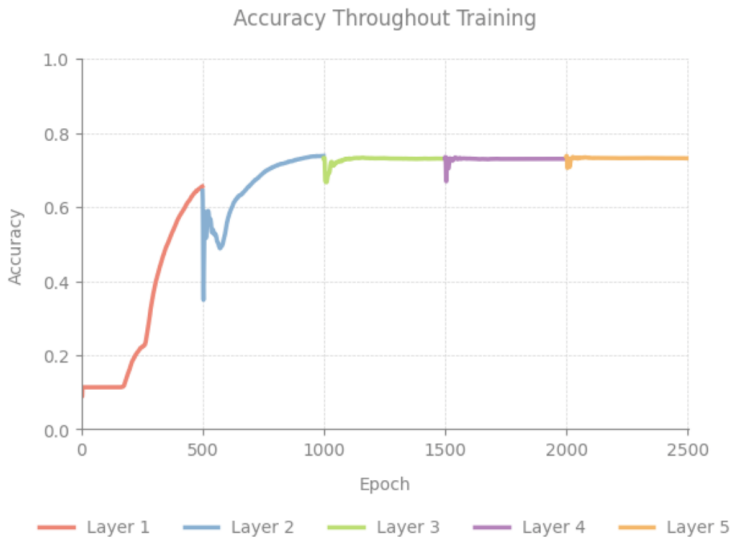


Figure 21: Chart depicting the learning curve of a multi-layer network composed of 5 layers, each with 500 neurons, trained over 500 epochs. The figure tracks the accuracy progression of the network throughout its training, illustrating the distinct phases of learning, including the rapid initial increase in accuracy and the subsequent stabilization.

The network's accuracy quickly increased before the addition of the second layer. However, when the second layer was introduced, accuracy was initially decreased as the newly untrained neurons began to be included in the classification calculation. Despite this initial drop, the accuracy recovered, surpassing that of the first hidden layer. A similar pattern was observed with the addition of the third layer: an initial drop in accuracy, followed by a recovery to the same level as before the layer was added. The same phenomenon occurred for the fourth and fifth layers. While all the layers were intentionally undertrained to make interactions easier to observe, most of the accuracy gains came from the first two layers, with the last three not contributing to improving the network's accuracy. Ultimately, the network achieved a training error of 78.92 percent and a test accuracy of 73.15 percent. This gap in accuracy is likely due to overfitting. To better understand how the network's predictive capabilities evolved during training, the heatmaps in Figure 22 compare the average goodness scores for each label.

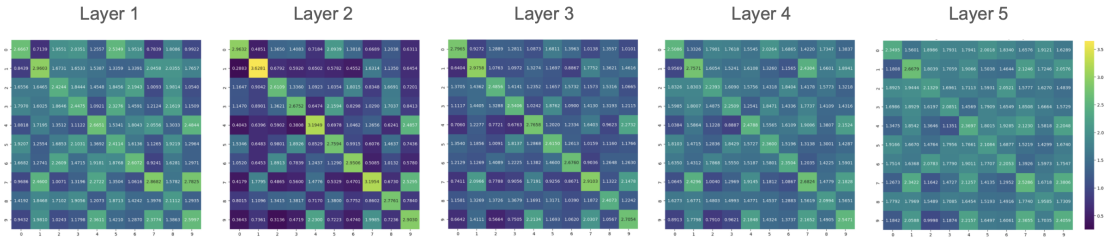


Figure 22: A series of heat maps illustrating the average goodness scores for each of the five layers in the multi-layer network, corresponding to the network configuration detailed in Figure 21. Each hidden layer consists of 500 neurons trained over 500 epochs. The heat maps provide a layer-by-layer breakdown, with each cell representing the average goodness score for a particular label tested against an input—the color spectrum from yellow to purple highlights the range from high to low goodness scores.

The heat maps display the average goodness scores for each layer in the network evaluated on the test data. Each cell represents the average goodness score of a label tested against an input. Yellow cells indicate higher goodness scores, while purple cells indicate lower ones. Each row is dedicated to a class of images, and the columns represent the labels tested against each image. Ideally, the greatest goodness scores would form a diagonal line from the upper left to the lower right, indicating that the highest goodness score corresponds to the correct label.

Observing the heat maps reveals interesting insights into how the network differentiates correctly labeled images from the nine incorrectly labeled ones. Starting with the first layer, the goodness scores appear somewhat random, albeit with a faint diagonal pattern. Despite the lack of clear differentiation between correct and incorrect labels at this stage, the network is already close to its peak performance, as seen in Figure 21. With the addition of the second layer, the diagonal pattern becomes more pronounced. When the goodness scores from the first two layers are combined, the network is very close to its peak performance. However, as the next three layers are added, although a visible diagonal persists, the distinction between the goodness scores for correct and incorrect labels diminishes with each additional layer. This diminishment leads to a stagnation in learning, as the network's predictions are based on the sum of the goodness scores produced by all five layers. While the last three layers do not detrimentally affect overall accuracy, their contribution to improvement is minimal. The diagonals in these layers contain the largest values, but the margin is small. Thus, instead of significantly increasing the total goodness score for the correct labels while minimally affecting the scores for incorrectly labeled images, these layers might as well add a nominal value to the total regardless of label correctness, given the minimal separation between correctly and incorrectly labeled images. This problem of layers not contributing underscores a central challenge in using multiple layers to calculate goodness scores. When additional layers are added, they need to excel in areas where

others do not. If all layers are proficient and deficient in the same aspects, there is limited benefit to adding additional layers.

4.4 Layer Interaction

After learning that adding more layers does not automatically result in better performance, I adopted a more strategic approach. To do this effectively, understanding the interplay between layers is crucial. My initial objective was to determine the impact of the neuron count in the first layer on the learning capability of subsequent layers. To investigate, I constructed three networks, each with a different number of neurons in the first layer, while keeping the neuron count in the second layer constant. The configurations of these networks are presented in Table 5 below, with the key differences between the networks highlighted in grey.

	Network 1		Network 2		Network 3	
	Layer 1	Layer 2	Layer 1	Layer 2	Layer 1	Layer 2
Neurons	10	100	100	100	1000	100
Epochs	200	700	200	700	200	700

Table 5: The configuration of three neural networks, each differing in the neuron count of the first hidden layer, with the second hidden layer's neuron count held constant. This table visually distinguishes the varying neuron counts in the first hidden layer, highlighted in grey, which is necessary to understand layer interaction and its impact on the network's learning capability.

Network 1 was configured with 10 neurons in the first hidden layer, Network 2 with 100, and Network 3 with 1,000 neurons in its initial hidden layer. The first hidden layer of each network underwent a training regimen of 200 epochs. After these initial epochs, the networks were still undertrained, making it easier for the second hidden layer to influence the network's accuracy. Subsequently, a second hidden layer consisting of 100 neurons was introduced to each network, which then continued training for an additional 700 epochs. The outcomes of this experiment are depicted in Figure 23.

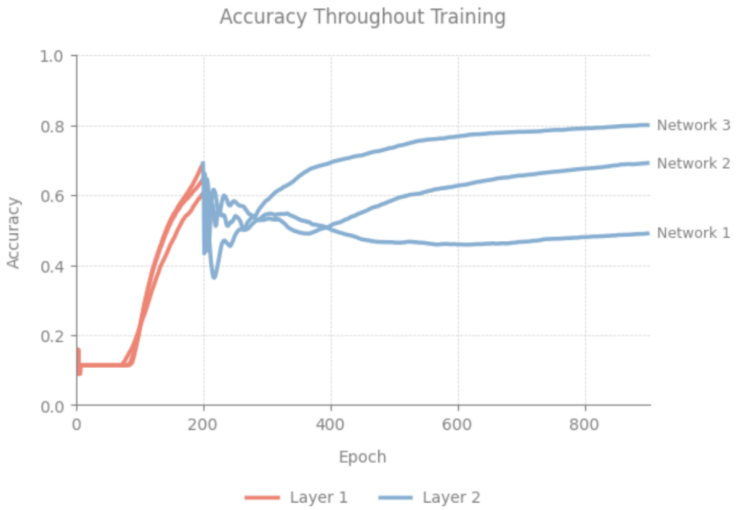


Figure 23: Plot of learning trajectories for three neural networks with varying neuron counts in the first hidden layer. It visualizes how the number of neurons in the initial layer impacts the overall learning process. This figure underscores the significance of a well-trained first hidden layer on the effectiveness of subsequent layers in a multi-layer network.

It is evident from the data that the neuron density in the initial layer plays a critical role in the network's learning trajectory. Specifically, Network 3, with 1000 neurons in the first hidden layer, shows a superior learning capability, reaching the highest accuracy after the addition of the second hidden layer. This indicates that a greater number of neurons in the first layer may contribute to a more robust feature extraction, which in turn enhances the overall learning potential of the network.

Conversely, Network 1, with only 10 neurons in the first hidden layer, demonstrates a considerable fluctuation in accuracy upon the introduction of the second hidden layer, suggesting that a sparse neuron count might lead to instability in learning when network complexity increases. Network 2, with an intermediate count of 100 neurons, representing a middle ground, achieving a balance between the rapid learning of Network 3 and the lower resource consumption of Network 1. It maintains a steady increase in accuracy without the volatility observed in Network 3, implying that there might be a threshold neuron count in the first layer that optimizes learning in subsequent layers without incurring the cost of diminishing returns.

The results underscore the importance of a strategic approach to neuron allocation in the first layer to establish a solid foundation for subsequent learning phases. They suggest that while additional neurons in the initial hidden layer can bolster the network's ability to learn, there is a nuanced balance that must be struck to prevent overfitting and ensure computational efficiency.

After examining the impact of the neuron count in the initial layer on the subsequent layers' learning capacity, my focus shifted to determining the optimal timing for introducing the second layer. Specifically,

the aim was to determine how much the first layer should be trained before commencing training on the second layer. To explore this question, three networks were constructed, each with 200 neurons in the first and second layers but differing training durations for the first layer. The configurations of these networks are presented in Table 6below, with the key differences between the networks highlighted in grey.

	Network 1		Network 2		Network 3	
	Layer 1	Layer 2	Layer 1	Layer 2	Layer 1	Layer 2
Neurons	200	200	200	200	200	200
Epochs	100	2900	200	2800	300	2700

Table 6: Showcase of training durations for the first hidden layer across three neural network configurations, each with an identical count of 200 neurons in both the first and second hidden layers. Key differences in the epochs dedicated to the first hidden layer training are highlighted in grey, offering a comparative look at how varying initial training durations can affect the subsequent layer's learning efficiency and the network's final performance.

Network 1's first hidden layer was trained for 100 epochs, Network 2 for 200 epochs, and Network 3 for 300 epochs. Following the completion of the first layer's training, the second layer was then trained until the total number of epochs for both layers reached 3000. This approach was designed to reveal the relationship between the initial layer's training duration and the overall effectiveness of subsequent training phases. The results can be seen in Figure 24.

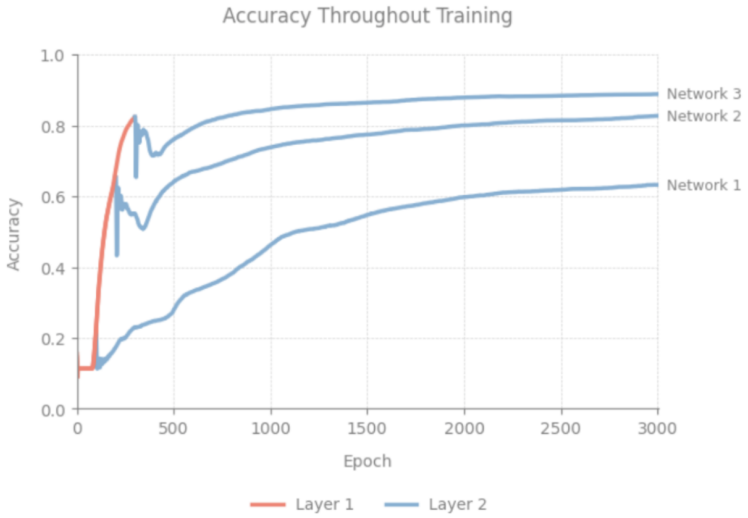


Figure 24: Plot of the performance for three neural networks that differ in the training duration of their first hidden layer. It highlights the network's learning progress and shows the relationship between the first hidden layers' training length and the following learning. The graph provides insight into finding an optimal training duration for the first layer that supports a productive learning progression without diminishing the value added by later layers.

The data suggest that the training duration of the first hidden layer is critical in determining the overall performance of a neural network. Longer training periods for the first hidden layer correlate with improved network performance. For example, Network 1, which received only 100 epochs of training for its first hidden layer, illustrates the harmful effects of an inadequately trained foundation. This shortfall in initial training restricts the second layer's capacity to learn, as reflected by Network 1's lower accuracy plateau relative to other networks.

Conversely, Network 3's first hidden layer underwent an extensive training period of 300 epochs, resulting in a considerable performance advantage. The improved training of this initial hidden layer provided a robust base that the second hidden layer, which exhibited the slowest learning rate, had a minimal impact on the overall accuracy. This indicates that when the first layer is extensively trained, it may diminish the learning influence of subsequent layers, having already established a significant portion of the feature detection and representation. Network 2 finds a middle ground, with a 200-epoch training duration for the first layer that is long enough to establish strong feature detection capabilities without eclipsing the contributions of the next layer. This balance ensures that the first layer forms a solid foundation for the second layer's learning. However, this network still does not perform as well as Network 1.

Contrary to initial appearances, the cap on total accuracy is only partially due to how the network's predictions are derived by summing goodness scores across all layers. Analysis of Network 1 in Figure 24 suggests that ignoring the goodness scores of the first hidden layer when calculating network accuracy could enhance performance; however, doing so actually results in a slight reduction in accuracy from 62.926 to 60.526. This outcome demonstrates that each subsequent layer builds upon the work of the previous one, thereby highlighting the benefit of optimizing each layer before introducing another. As observed in earlier experiments, the first layer tends to experience a swift increase in accuracy before reaching a plateau. Therefore, pinpointing the optimal training duration for the first layer depends on the specific problem at hand, the task's complexity, and the neural network's architecture.

4.4.1 Weighted Layers

A crucial aspect of multi-layer networks appears to be controlling the input from each layer in the final prediction. One way to control the influence each layer has is by using a weight multiplier for each layer. To investigate the implementation of a multiplier, I need to construct a network where the accuracy at the end of the first layer's training is lower than that of the second layer. To achieve this, a network with two hidden layers, each comprising 500 neurons, was created. The first hidden layer was intentionally undertrained to foster some distinction between the layers' contributions deliberately. The second layer

underwent an extensive training period, totaling 2000 epochs. The test accuracy tracked during the training duration is illustrated in Figure 25 below.

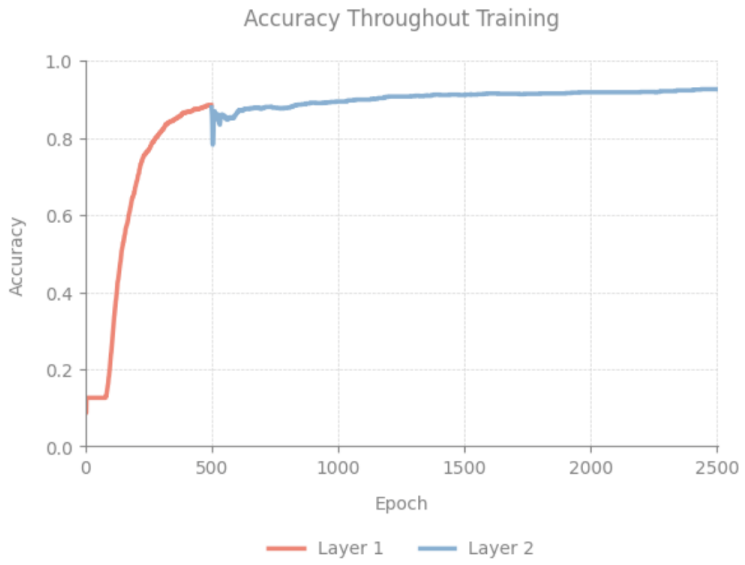


Figure 25: Chart depicting the accuracy of a two-layer neural network during training, where the first hidden layer, containing 500 neurons, was trained for 500 epochs, establishing a foundational level of learning. Subsequently, the second hidden layer with 500 neurons underwent a more extended training of 2000 epochs.

As expected, the first layer exhibited significant initial improvement, and just as it was approaching a plateau, its training was halted, shifting the focus to the second hidden layer. Upon the second layer's training commencing, there was an initial drop in accuracy, followed by a gradual improvement. The completed network achieved a training accuracy of 92.45 percent and a test accuracy of 92.60 percent. Normally, the goodness scores of the first and second layers are combined to calculate the network's overall goodness score for image classification. However, the weights could be applied to the goodness scores of each layer before they are combined to make a prediction. This approach could serve as a way to refine the network's decision-making process by emphasizing the more reliable layer's output in the final prediction.

Examining the plot in Figure 25, it becomes evident that the accuracy of predictions improves after adding the second hidden layer. With this knowledge, I began experimenting with the network's weights by setting the first layer's multiplier to 0 and the second to 1. This adjustment resulted in the network making predictions solely based on the second layer, which interestingly caused accuracy to fall to 91.8 percent. Conversely, if the second layer's multiplier is set to zero and the first to one, accuracy drops further to 88.4 percent. These results are intriguing because one might expect the second hidden layer alone to perform better than the complete network, considering the second hidden layer is essentially a refined version of the first. However, the decrease in accuracy indicates that the first layer captures certain aspects of the input

that the second does not. While zeroing out one layer led to reduced accuracy, there may be a combination of weights that could vary the influence of the layers in a way that improves accuracy. To visually examine how varying the multipliers on the goodness scores for each layer affects the network, the following heatmap in Figure 26 was created, with the y-axis representing the first layer's multiplier and the x-axis representing the second.

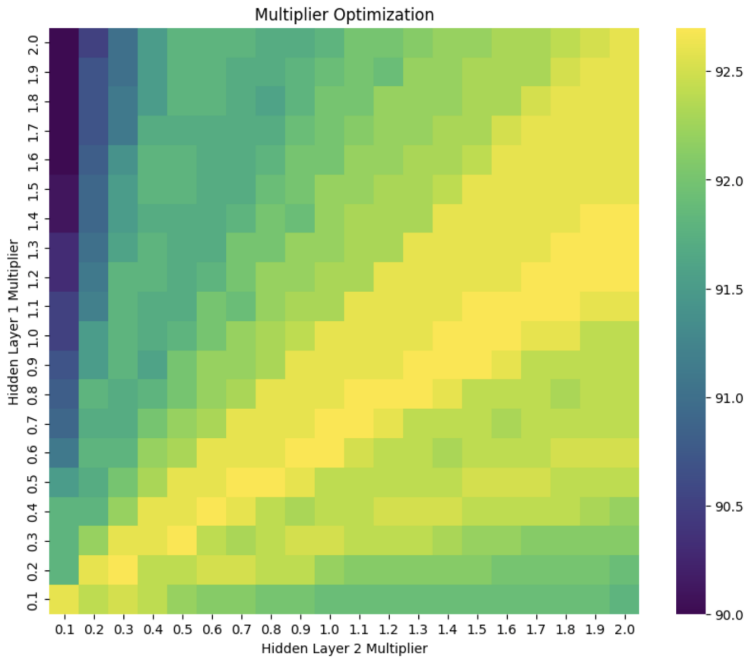


Figure 26: A heatmap depicting the impact of applying different weight multipliers to a network's first and second hidden layers on the test accuracy. The y-axis represents the multiplier for the first layer, while the x-axis corresponds to the second layer's multiplier. The color gradient from yellow to purple indicates higher to lower accuracy scores, respectively.

In Figure 26 the heatmap illustrates the impact of changing multipliers associated with the first and second layers on the test accuracy of the network. Yellow represents higher accuracy scores, and purple represents lower accuracy scores. As the heatmap indicates, the optimal performance comes from a balance where the ratio of multipliers leans towards a value around 0.6. Deviating from this ratio and altering the ratio of influence leads to reduced accuracy. Applying a multiplier of 1 to both layers results in an accuracy of 92.60 percent, which is not the best possible performance for the already trained network. However, a ratio of approximately 2 to 3 between the first and second layers slightly improves accuracy to 92.70 percent. Although this represents a modest increase, it demonstrates the potential of multipliers to fine-tune network performance.

5 Conclusion

This project's primary goal was to evaluate the FF algorithm as an alternative to gradient descent with backpropagation. Based on the results, the FF algorithm, while not without imperfections, is capable of effectively training a neural network in a layer-wise fashion by locally updating the weights and biases. It adopts a unique approach that seeks to maximize the goodness of positive data while minimizing that of negative data. In doing so, the network learns to differentiate better between images that are correctly labeled and those that are not. Thus, when it comes time to make a prediction, the network can test every label to a given input, and the label that produces the greatest goodness score is selected as the network's prediction.

5.1 Key Takeaways

This project demonstrated that the Forward-Forward (FF) algorithm can effectively train a neural network. Although everything from its inputs to the approach it takes to train the network and the interpretation of the network outputs radically differs from traditional training processes, the network can still distinguish positive and negative data. The network used this ability to accurately classify images in the MNIST data set. The classification strengths and weaknesses of the network are consistent with expectations, and misclassifications typically occur in both directions. The network demonstrated a strong ability to learn very quickly, realizing most of its accuracy within a few hundred epochs; however, after this, learning plateaus and accuracy gains slow down dramatically. This phenomenon may result from the simplicity of the MNIST data set. However, the local loss function is much simpler than a global one, which may contribute to the smooth and fast training experienced. It's important to note that the network's classification abilities fell when noise was added to the images. This fall indicates that this simple implementation of the FF algorithm demonstrated in this project may need to be revised for more complex datasets.

Another interesting finding was that the FF algorithm performed surprisingly well on networks with a limited number of neurons in the first hidden layer. For single-layer networks, the difference in performance between 10 neurons and 1000 neurons in the first hidden layer was minimal, which again may be a result of the simplicity of the MNIST dataset. With or without noise added to the images, it was observed that the number of neurons in the first hidden layer is directly related to the network performance; as the number of neurons increases, the potential accuracy of the network increases at a decreasing rate. When a second layer is added to the network, the number of neurons and epochs of previous layers trained influence the

subsequent layer's ability to learn. This influence is because the previous layer's output is used to train the next layer. As a result, this project showed that fewer neurons in the first hidden layer mean less data is passed, reducing the second hidden layer's ability to learn. The number of epochs in the first hidden layer also impacts the second hidden layer's ability to learn. With more epochs, the data quality passed to the second hidden layer improves, resulting in an increased ability to learn.

This project also revealed that if the outputs of several layers are similar, adding their goodness together to make predictions has little benefit. Based on the findings of this project, several layers trained with the same loss will produce similar goodness scores for the same input. This result doesn't mean that subsequent layers aren't learning; they are not learning anything that the first layer isn't. Unfortunately, if each layer assigns a similar goodness score, adding more layers will universally increase the network's goodness in an untargeted way. Based on this phenomenon of subsequent layers learning the same thing, it would be better for different layers to learn different things.

Perhaps the most exciting takeaway from this project is that the goodness scores of each layer can be manipulated to enhance performance. To control each layer's influence, a multiplier can be applied to change the goodness scores of each layer. Then, when the total goodness score is calculated to make predictions, the multipliers will determine the influence of each layer's contribution to the total goodness. While this project provided a conceptual demonstration using a heatmap to identify optimal multipliers visually, adding a multiplier to the goodness score of every layer is a straightforward way to adjust the influence of each layer. However, more complex approaches that scale the goodness scores could be implemented relatively easily. In addition to changing the scaling method, a more practical solution for determining the most effective method might involve using an optimization algorithm like gradient descent.

5.2 Future Work

While the development of the Forward Forward (FF) algorithm is in its early stages, there are several directions future research could take. One of the most promising developments could be creating a custom loss function for each layer. In my implementation, using the same loss function for every layer resulted in each layer being good and bad at the same things, offering minimal benefit from adding additional layers. One potential loss function variation could involve alternating training between layers that maximize goodness for positive data and minimize it for negative data with those that do the opposite. In addition to this approach, there are numerous other methods that could be implemented to attempt to create layers that complement each other.

Another avenue worth exploring is finding the best approach to control the influence of individual layers on the total goodness score. As more sophisticated loss functions are developed, there will be a need to standardize the goodness of each layer. Using the same loss function on all layers ensures no single layer dominates another. However, with varying loss functions producing goodness scores differently, combining them without manipulation is unlikely to yield the best outcome. Instead, they must be adjusted, which could be done linearly, with an exponent, or with a function. It will also be essential to optimize the weights for each layer. The optimization of weights could be done with backpropagation or another similar optimization algorithm.

Another avenue worth exploring is the simultaneous training of all layers. In my implementation, each layer was trained for a prescribed number of epochs before training ceased and moved on to the next layer, which was then trained for a set number of epochs. It's challenging to predict what the results of this approach would be since the second layer would be trained using the output of an in-training first layer instead of a fully trained first layer. Additional directions for future research could include training all layers simultaneously instead of fully training one layer before moving on to the next, experimenting with different datasets that provide more of a challenge, exploring alternative methods for generating negative data, and investigating different labeling techniques and patterns.

References

[1] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," *arXiv preprint arXiv:1811.03378*, 2018.

[2] J. Schmidhuber, "Annotated history of modern ai and deep learning," 2022.

[3] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[4] G. Hinton, "The forward-forward algorithm: Some preliminary investigations," 2022.

[5] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton, "Backpropagation and the brain," *Nature Reviews Neuroscience*, vol. 21, no. 6, pp. 335–346, 2020.

[6] J. D. Olden and D. A. Jackson, "Illuminating the "black box": a randomization approach for understanding variable contributions in artificial neural networks," *Ecological modelling*, vol. 154, no. 1-2, pp. 135–150, 2002.

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[8] J. Pearl, *Causality*. Cambridge university press, 2009.

[9] T. Hastie, R. Tibshirani, J. H. Friedman, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction*, vol. 2. Springer, 2009.

[10] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[11] M. A. Nielsen, *Neural networks and deep learning*, vol. 25. Determination press San Francisco, CA, USA, 2015.

[12] H. Anton and C. Rorres, *Elementary linear algebra: applications version*. John Wiley & Sons, 2013.

[13] F. Chollet, *Deep learning with Python*. Simon and Schuster, 2021.

[14] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[15] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

[16] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, JMLR Workshop and Conference Proceedings, 2010.

[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[18] C. M. Bishop, *Neural networks for pattern recognition*. Oxford university press, 1995.

[19] M. Spivak, *Calculus*. Cambridge University Press, 2006.

[20] L. N. Smith, "Cyclical learning rates for training neural networks," in *2017 IEEE winter conference on applications of computer vision (WACV)*, pp. 464–472, IEEE, 2017.

[21] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[22] S. Raschka, *Python machine learning*. Packt publishing ltd, 2015.

[23] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[25] P. Mohammad, "Pytorch frward-forward," 2023.

[26] T. Salimans and D. P. Kingma, "Weight normalization: A simple reparameterization to accelerate training of deep neural networks," *Advances in neural information processing systems*, vol. 29, 2016.