How can different neural network starting models be used to improve training speed?

Wells Wait

Senior Project Advisor: Kyle Edmondson

Abstract

Artificial intelligence and machine learning are becoming extremely prevalent in modern society, but it takes significant computational (and electrical) power to train these systems. Big companies have the resources to train these large models, but the amount of power affects the grid and limits the speed at which models can be implemented. This paper examines different neural network starting models and how they might improve training speed. Several classification models and regression problems were tested, and both their error rates and operation speeds were compared to assess each model's performance. It was determined that there was no ‘best’ starting model for every situation, meaning that time needs to be spent identifying proper starting models to achieve best performance in a given situation. In the future, more studies should be done creating detailed systems for starting model selection.

12th Grade Humanities
Animas High School
11 March 2025

Part I: Introduction

The algorithms that cause fascination and fear in our modern society, that control the information we receive and use to learn the complicated interactions of the world, are really bad at learning. AI, a topic that has taken this world by storm several times, is surprisingly inefficient and ineffective at the very things it is designed to utilize. AI, properly called artificial intelligence, is a subfield of data science, which contains the subfield of machine learning, which is often used as an interchangeable term with AI. Machine learning is the field of study of making computers learn patterns in data. Patterns in data can be used to describe the entire world around us, from the basics of Newtonian physics to modern relativity, and even art, and if these patterns can be modeled and predicted, our computers can master the world around us. But luckily for those who are afraid of their laptops taking over the world, there is a significant challenge in finding and modeling the patterns of the world. Even with our most advanced computers and large amounts of power, we can only barely model human language. If you learn a pattern, it is easy to see what comes next, but to learn a pattern takes lots of effort and is currently the main challenge in machine learning and artificial intelligence. ChatGPT3 took 34 days and 1064 MWh of power to train, and that is nothing compared to the more modern version 4 (Zodhya). This major power and time commitment, in addition to the hardware and data necessary, has forced AI to be something only big companies can truly utilize to a large extent. A lot of research goes into improving hardware and, more recently, training algorithms to decrease this cost and improve the larger models' training speed and cost effectiveness, which is extremely important. While some people may be scared of change, these algorithms do nothing more than predict outputs that match a pattern, and improving these starting models is critical to societal advancement and a better understanding of our own ability to learn. There are many possible

areas of AI improvement, but the focus of this paper is on the basic Neural Network and the starting model, as it is the basis for most everything in the field. Different starting models result in different distances to an optimal model and varying computational costs per situation, resulting in the fastest starting model being extremely situational.

2 Part II: Historical Context

The idea of modeling the human brain with computers has been around for a very long time. A report on AI from Stanford describes the first time a computer tried to model a brain, which occurred in 1943, where they were identifying how the brain operates. As a part of this research into the brain, they "modeled a simple neural network using electrical circuits" (Myszewski, Dave, et al). The neural network has evolved much since this time of a simple model of a few neurons. However, the artificial neural network is still built around how the brain operates. Several newer algorithms use more modern studies on the brain as inspiration, like the spiking-neural network (Horak). In 1959, two models were developed by Bernard Widrow and Marcian Hoff, one of which was called MADALINE, which "was the first neural network applied to a real world problem, using an adaptive filter that eliminates echoes on phone lines." This method of improving phone audio quality is still in use to this day, despite the fact that soon after this implementation the idea of AI was temporarily left to the side of the road due to limitations (Myszewski, Dave, et al).

The early implementations of AI were impressive but doomed to failure due to the lack of development of circuits, transistors, and other modern technologies, but it might be possible now (Myszewski, Dave, et al). Our computers and our understanding of circuitry are far above that of the 20th century. With these advancements, our neural network structure can now be large enough to model complex problems. While we have the computers to run more advanced

models, we are still struggling to keep up with the problems we want to solve, such as human speech. As computers develop and are designed to better work with the neural network structures, we will be able to run more complex simulations, but developing better mathematical structures can also allow this to happen. The mathematical structures allow these models to work but have not been improved as much as computers.

3 Part III: Background

3.1 Overview

This section will provide a conceptual overview of what machine learning is and how neural networks operate and train. This section will also introduce terminology around the neural network structure that will be needed to understand the later sections. This section will be the conceptual basis needed for understanding the comparisons in later sections.

3.2 Machine Learning

Machine Learning is a subfield of data science with a focus on identifying patterns and using these patterns to predict outputs. This is used in many areas, with the simplest application being linear regression. This idea of fitting a line is generally not enough, as most patterns are not linear, thus requiring fitting nonlinear functions. This, it turns out, can be very difficult and is called curve fitting. Curve fitting is the process of attempting to match a curve to a set of data points by adjusting parameters like slope and concavity. By adjusting these parameters, you change the error, sometimes called loss, of your function compared to the curve. This is often

calculated by finding the distance from the data to the curve and then squaring it to ensure it is positive. This gives a metric that can be used to identify how well you fit data.

A demonstration of this can be seen in Figure 3.2.1, which shows a polynomial fitting some data. The polynomial that is attempting to fit the data is a simple quadratic $y = ax^2 + bx + c$, with two weights and one bias. A weight is an adjustable coefficient of the input, and a bias is an added constant. These are adjusted during the fitting process and change the curve to closely match the data, minimizing the Error function.

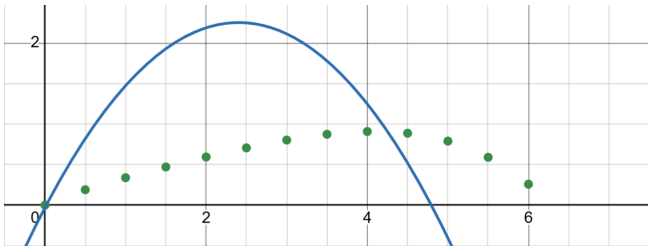
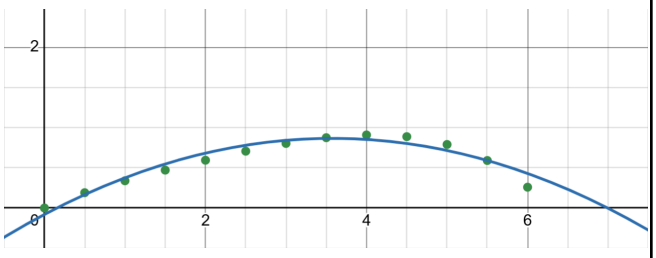
Before Fitting	After Fitting (using quadratic)
	

Figure 3.2.1 shows an attempt at curve-fitting data with a quadratic

All types of machine learning work on this idea or something very similar. In this case, we are discussing supervised learning where the error function is based on known values. This is not always the case in reinforcement learning, where the Error function is based on the environment. The most easily used type of machine learning is supervised and is what everything in the paper will be based on, including the curve fitting.

3.3 Network Style Models

Machine learning is a very large field with many different approaches, but one of the most well-known is the Neural Network (NN), which is the focus of this paper. A NN is a structure that is designed around maximizing the number of unique tunable parameters, allowing it to be able to fit data better. In the example with the quadratic in section 3.2, there were three adjustable parameters allowing for parabolic and linear curves to be easily fit, but most patterns are not this simple. As the tunable parameters change how an equation looks, having more adjustable values allows for more types of curves, allowing them to be utilized for complex patterns.

While it may seem a good idea to just increase the number of parameters as much as possible, there are some issues with this. This paper will not go into depth on all the issues behind this, but the two big issues are overfitting and computational cost. With more weights, you can better complicate patterns, but if you have too many parameters, you can fit the data while missing the pattern. Take Figure 3.3.1, while this data is decently fit, it clearly does not find the actual pattern, which is just a simple quadratic. The other issue with adding more parameters is the time it takes to train or learn the pattern. In Figure 3.3.1, it is clear that if this is a polynomial trying to fit the data, it probably has a probably a 100th order polynomial, which is hard to fit. If you try, you can probably fit a simple polynomial by guessing and checking, as there are three terms, but with 100 weights, it would take a lot of time to properly identify a curve. This all just goes to say that adding more parameters is only sometimes good.

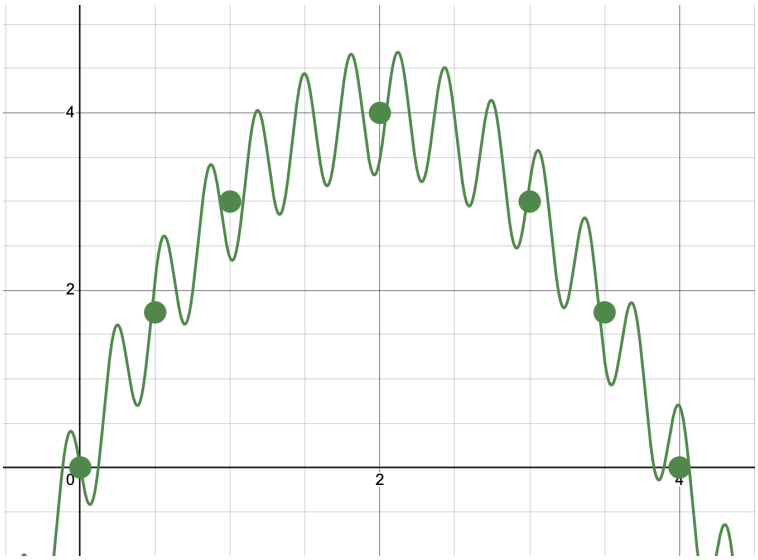


Figure 3.3.1 Overfit curve fitting example

While NN model is used to add more adjustable parameters, it was initially designed to model the human brain. This means that there are neurons and connections between them, which are used to build the equation. Each neuron is a small function that takes some number of inputs and outputs one value. Figure 3.3.2 shows this happening with each of the neurons(circles) in the first layer (green box) having one color of output. This represents that each neuron has one value of output, but as can be seen with the pink neurons, there are multiple inputs to each neuron. The way this type of model works is by first inputting values into the first layer. Each neuron in the first layer runs its calculations and then outputs a value to each neuron in the next layer, with the last layer's outputs being the output of the entire network.

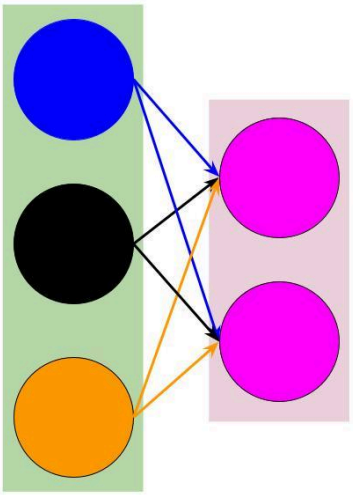


Figure 3.3.2 shows a simple neural network structure

The method described is used only in dense neural networks called multilayer perceptrons (MLP). This method is not the only way, but it is extremely common. All of the neural network structures have some things in common, like the fact that they operate layer by layer, meaning one layer is computed at a time. The differences in the structure are around the connections. For example, you do not need every neuron in the next layer to get all the outputs of the previous layer. This type of modification decreases the math that is computed because there are fewer inputs.

The operation of the MLP style of model is fairly simple, and while it does not look like a function, it is. This structure of neurons can be written all out in one equation (per output), with the neuron outputs being the input to the next layers of neurons, resulting in a large multi-composite function. This function written out is complex and hard to work with, making it generally ignored.

3.4 Model

The model is what you call all the weights, biases, and structures, which allow the model to give predictions. In other words, the model is the function, which includes how the function is laid out and the components. There are four major comments of the model: the architecture, the neuron equation, the activation function, and the adjustable parameters. Each of these come together to form how the model takes an input(s) and delivers an output that predicts what would happen in the real world given that input.

The architecture of a model is how the data flows between neurons and where these neurons are located relative to each other. An example architecture is shown in Figure 3.3.2, which shows a simple two-layer architecture with three neurons in the first layer and two neurons in the second. This figure also shows how each neuron in the first layer gives its output to every neuron in the second layer. This is an example of an architecture as it defines where and how each neuron is connected. One aspect that is not shown in figure 3.3.2 is the input size, which, as undefined, could be anything; this is normally included in an architecture and is also normally shown in figures.

The neuron equation defines how each neuron combines and modifies each input into one output. This is the main topic being discussed later in this paper, as the neuron equation is the actual math that builds up the function, while the architecture is how these equations combine into one. The most basic neuron equation multiplies each input times its respective weight, and then adds them together with a constant. This weighted sum is then put through an activation function.

The activation function is a simple nonlinear function that allows a mostly linear neuron equation to duplicate curvature found in most real-world data. Most neuron equations are just

inputs times a coefficient, making them linear, but as most real-world situations have nonlinear aspects, the activation function is used to add this feature into these neurons and the overall network. The activation function was initially just a step function representing how human neurons are either on or off, but this has been changed to a continuous function that often still has similarities to the step function but is more complex. Some common activation functions are sigmoid and ReLU, with sigmoid modeling a curved step function and ReLU being a mostly linear function. While these functions add nonlinearity, they are unable to rely on the adjustable parameters to utilize the nonlinearity.

The adjustable parameters that are adjusted during training are called weights and biases. These have been referenced previously, and simply either multiply or add to an input. These parameters are adjusted to better match the patterns in the data during training, but are often started randomly with small values as there is no easy way to know where they will end up.

The starting model is where all of these factors start before training. Initial weights and biases are often set randomly, and the structure is estimated based on the complexity of data and activation functions based on personal preferences and data characteristics. During training, the weights and biases are slowly adjusted based on the starting values, making the starting model affect training speed. This can be thought of as how far off your first guess might be when curve fitting, if your guess is a sine function, but what you need is a third-order polynomial, it will take a lot of work to get it to match. However, if you guess a third-order polynomial to start, all you will have to do is slightly adjust the parameters to match. This is the same idea behind picking a good starting model, but this can be very difficult. In this paper, we are examining this by picking neuron equations.

3.5 Training

While the focus of this paper is to examine the starting model through the neuron equation, this can not be done without understanding how training adjusts the starting model's parameters. While we as humans can see and identify patterns without much thought in three dimensions, duplicating this in math and in more dimensions can be difficult. There are several different training algorithms that can do this, but the most common by far is backpropagation with sparse gradient descent, which is what will be used in this paper. It is by far the most common and operates in a relatively intuitive way.

Backpropagation is a process of identifying how each layer should change its weights and biases, starting at the output layer and working backward to the input. The training starts at the output layer as the error function is based on a comparison of this layer's output and the correct output. Then, based on how the output layer changes, and specifically how the inputs to the layer should be changed, the next layer can be updated. This progress then continues through all the layers, ending at the input layer.

The first step in backpropagation is calculating the error at each of the output neurons, which is done by running an input through the model and calculating the difference between the model's output and the desired output, and squaring it. This gives us the amount needed to adjust its outputs, which can be used to identify how the weights, biases, and inputs should be adjusted. This adjustment to the inputs is used to identify the next layer adjustment instead of the error, as we can not get the error partway through the network. This can also be thought of as working from the outside of a composite function to the inside, which is actually what is happening. But while it makes sense to move backward through the model, this does not explain how to adjust the weights and biases.

Gradient descent uses the gradient at a point to identify how to adjust the weights and biases to decrease the error, also referred to as cost. The gradient is a vector that points in the direction of the greatest change, and if calculated for our error function, we get the direction of the greatest change to the error based on the weights and biases of that layer. This is the same as finding the greatest slope in multiple dimensions, with weights and biases being our independent variables. If we calculate and average the gradient of all the data points, it will show how to increase the cost the fastest. By adjusting the weights and bias in the opposite direction of the gradient vector, it will decrease the cost in the fastest direction. Now, this may seem unreasonable when working with hundreds of thousands of data points as that is a lot of computation for one adjustment. Because this is so computationally intense, it is common to use sparse gradient descent for most models.

Sparse gradient descent takes a subset of the data, called a batch, and runs gradient descent on it. This does not necessarily give the best way to adjust the weight and biases, but the adjustment is good enough to improve the error, and the increase in operation speed makes up for it. Sparse gradient descent is not always used when working with small datasets but is necessary when training larger models.

Training is how the model learns the pattern and is critical, but it also takes a lot of energy and time. It is the real issue when it comes to modern models and is the focus of this paper's investigation. The training speed is affected by a lot of factors, including the neuron equation. The next section goes over several different neuron equations and the math it takes to train and operate the model before comparing them in a series of tests.

4 Part IV: Summary of Past Research and Analysis

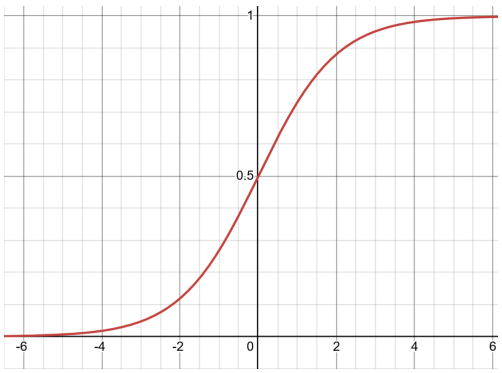
This section will go over the math and ideas behind several different attempts to improve the starting model. Starting with an examination of common activation functions' effect on neuron equations, followed by several different neuron equations and architecture styles that attempt to improve some aspect of the Neural Network style of machine learning

4.1 Activation Functions

The activation function is the most common way to adjust the starting model and various alternatives are compared in the testing section. As previously discussed, the activation function is a function that adds nonlinearity and often controls the size of the neuron outputs. They are also commonly adjusted due to the fact that they change a lot about the layers neuron equation without requiring much time to adjust. There are a wide variety of activation functions, and though not the primary focus of this paper, this section covers a few of the most common approaches.

Information regarding the activation function comes from 3blue1brown and a general understanding of math (Sanderson). This section is more of a summary of every source and personal experience relating or talking about activation functions. This section should be used to understand reference to activation functions in the later section, but not used as a foundation for understanding activation functions or error functions. This paper is not about the activation function as much as architecture and thus a general understanding is enough to grasp the concepts.

4.1.1 Sigmoid:

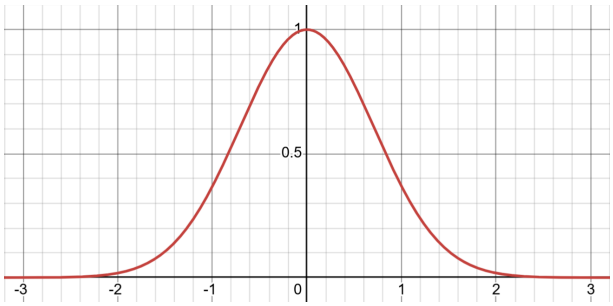
Function:	Graph:
$f(x)=\frac{1}{1+e^{-x}}$	

The sigmoid activation function is an extremely common, if not the most common, activation function. This function has some resemblance to the Step function that was used to better model the human brain and generally forces the output to be 0 or 1 as only a small section of the domain gives other values. This bounding of the function also slots in smaller output values, which can increase computational speed. In addition to the sigmoid function being bounded, it is injective, with each input having a unique output, which is a benefit as it avoids treating two outputs similarly unless they are similar. With all these benefits, it is no wonder people tend to use this function, but while it is good in most scenarios, this function's horizontal asymptotes can cause issues during training.

During training, the use of the sigmoid derivative does cause slower training speeds. The steepest slope is at 0, with the derivative getting smaller as you extend to either side, with the values getting close to 0 rapidly. This causes an issue during initial training where values imputed into the function could be extremely large, resulting in a derivative close to 0. In a practical sense, this means that at that point, the gradient vector is small enough to basically result in no adjustment to the weights or biases. In addition, most models that are classifying things have 0 and 1 as desired output, meaning that the model trains that the derivatives will get

smaller. This is not a bad thing necessarily, but this sigmoid function takes it to an extreme. This means that the sigmoid activation function often results in slow training.

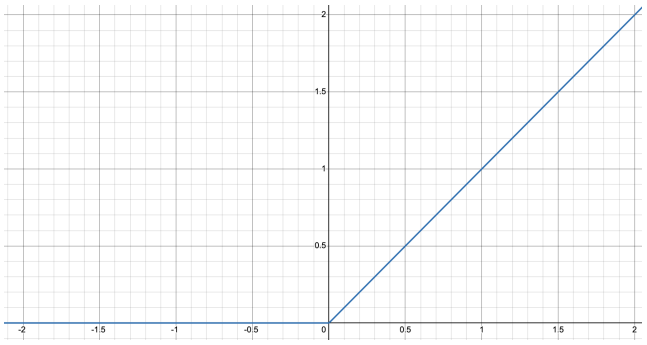
4.1.2 Gaussian:

Function:	Graph:
$f(x)=e^{-x^2}$ Sometimes written: $f(x)=\frac{e^{-x^2}}{2}$ This can make the derivative simpler to compute.	

The activation function looks very different from the sigmoid function but does have some of the same characteristics. It is considered to be more computationally intensive and is not injective, meaning multiple inputs give the same output. This makes this function less commonly used. While it is less commonly used, it still has its benefits, mainly in that it zeros value on either side of 0. This means that it is about getting the right value to this activation function, unlike the sigmoid activation function, which will give an output as long as you have a large enough output. This is often not enough to make people choose this function, especially as it still has two horizontal asymptotes, slowing down training.

This function has similar training characteristics to the sigmoid with two asymptotes and relatively small vertical bounds on the derivative. The one difference is that the symptoms are both going to 0. This means that if, during training, one of your targets is the value 1, you will likely have faster training for longer as the derivative near this desired output is still relatively large compared to that of the sigmoid function. While this function is not as common as the sigmoid, it is still popular and useful for specific tasks.

4.1.3 ReLU:

Function:	Graph:
If 0<x: f(x)=x If 0>x: f(x)=0	

This activation function is almost linear, meaning that it is not as much use for finding nonlinear patterns but the linear nature makes it fast and easy to compute. During normal operation, this function is extremely fast, only changing values smaller than 0 to 0 and leaving the rest identical to how they were inputted. This makes this model almost as fast as just a linear activation function. This function's speed makes it a common pick, especially before a more nonlinear activation function to speed up computation without losing all nonlinearity. This function does output larger values, resulting in the possibility for slower computation in the case of extremely large values, but so long as the inputs are not overly large, it only speeds up the process. The cut-out of values smaller than zero means you want to operate with positive values in your network as much as possible to avoid the loss of data.

This model is unable to fit nonlinear data very well, but the derivatives are simple to compute, making it train fast. The derivative is either 0 or 1, depending on whether the input is negative or positive. This means it either prevents change or does not affect the amount of change being applied to the weights and biases. This does not affect training speed much at all.

As this particular function is not linear, it is often paired with a nonlinear function on the next layer, like a sigmoid. This helps keep the model running fast while keeping most of its ability to learn nonlinear data on simple problems.

4.2 Basic Neural Network

The Basic NN is the most common neural network structure and what most of the models discussed in this paper are based on. This neural network can model any function given enough neurons and training time, which is why this is used to fit unknown functions. A simple logical proof is given by Derrick and Widrow. Take any function and subdivide it into infinite linear pieces, then build a neural network where there is a neuron per section of the function. Then, each neuron can handle a single piece of the function and thus model the function if the sections are small enough (Nguyen and Widrow). This logical explanation makes sense, and there are other papers that go into more depth, but this characteristic is why this model is so popular.

While this most basic structure is commonly used, it is not the best for every situation and other models with different math might be better. While most of the models in this paper are built off the math of this function, slight variations can allow for different benefits and downsides that, depending on the situation, are beneficial.

This section explains both how the Basic NN operates as well as gives the foundations for the other models. This section explains the math in depth, as this is the most basic model and can be used to understand the rest of the models. The training is given in two methods, one is much simpler to understand with little background, and is the main method of explanation for other models. There are a few models that can only be explained through matrices of partial derivatives. This is the second method shown in this section, and it allows you to see how the

simple and matrix versions of training are connected, hopefully allowing an understanding of training in some of the more experimental models.

This section is based on a video series by 3blue1brown, a mathematician, explaining the inner workings of neural network operation and their training. Some of the math sections are extended beyond the contents of the video to help explain other models, mainly through matrices of partial derivatives.

4.2.1 Neuron Equation

The neuron equation determines how the inputs to a neuron are processed and combined into one output. The inputs and outputs can be any value, but are normally both scaled to values between 0 and 1. The neuron equation has to take these values and combine them in a way that can pull apart the input and give the desired output. The below is the neuron equation for the Basic Neural Network and the foundation for most other neural networks.

The basic Neuron Equation:

$$\sigma\left(\sum_{i=0}^{N} w_i x_i + b\right)$$

Where σ is the activation function, W_i is the weight for input i, x_i is input i, b is a singular bias per neuron, with N being the number of inputs.

This equation is fairly simple, and a computer can compute these types of equations extremely quickly. The summation adds all the inputs after first being multiplied by a weight, with each input having its own weight per neuron. This part of the equation is called the weighted sum, and after done, a bias is added to handle offsets. The last step is using this sum in

the activation function to add nonlinearity before sending the output to the next layer's neurons. This equation only models one neuron, however, needs to be done k times with k being the number of neurons, with each neuron taking the same inputs but using different weights and biases. Computers use matrix math to handle all the neurons at the same time, increasing the speed. But all of these weights and biases shown in the equation must be adjusted to properly duplicate patterns, which happens during training.

4.2.2 Simplified Training

The training of these neural networks is conceptually explained in the background section; however, this discussion is inadequate when comparing how the models are to be trained through math. This section covers the simplest way the math can be represented for training of this network style. This method does not always work, but when it does, it allows for much easier understanding of the computation and issues that can occur during training. The simple and more complicated method starts the same with defining the error of the outputs, which is also called the cost in this section.

Cost Equation:

$$C = (Y - P)^2$$

Where C is cost, Y is the true value, P is the model prediction.

This cost function is defined with P, which is the output of the neuron being trained. This is only done at the output layer as there is no true value at other layers. Instead of a cost function, they use the partial derivatives of the previous layer in terms of the inputs. In other words they change the output of non-output layers. This allows for training of multiple-layer training despite the lack of true value.

This cost function is what we take derivatives of to get the gradient, vector of greatest change, in terms of weights, biases, and inputs (to train non-output layers). To calculate the gradient, the cost function's derivative must be calculated. Below is a composite function separated to allow for derivative calculation. Note that there is a summation in equation Z, this summation can be ignored when taking Z's derivative, as each of the input and its corresponding weights are independent between indexes. This means that as we need the derivative in terms of each weight and input, we can take a derivative for each index and ignore the summation. This results in getting the partial derivative in terms of every weight and bias.

Separating the Equation:

$$C = (Y - P)^2$$

$$P = \sigma(Z)$$

$$Z = \sum_{i=0}^{N} w_i x_i + b$$

Derivative of the cost function in terms of P:

$$\frac{dC}{dP} = -2(Y - P)$$

This can be read as the change in cost per change in prediction, P, which is what we want to know.

Derivative of P in terms of Z:

$$\frac{dP}{dZ} = \sigma'(Z)$$

Were σ' is the derivative of the activation function

Derivative of Z in terms of w,x, and b of just one index of the summation:

$\frac{dZ}{dw_i} = x_i$	$\frac{dZ}{dx} = w_i$	$\frac{dZ}{db} = 1$

With all the derivatives computed, they can be combined to get DC/dw, dC/dx, and DC/Db or the change in cost per change in the weights, biases, and inputs. To get the final result for cost in terms of the weight, bias, and inputs requires multiplying all the derivatives together as shown below. This is the same as the chain rule and allows us to find how one input and its corresponding weight needs to change to improve the cost.

$\frac{dC}{dw_i} = \frac{dC}{dP} * \frac{dP}{dZ} * \frac{dZ}{dw_i}$	$\frac{dC}{dx_i} = \frac{dC}{dP} * \frac{dP}{dZ} * \frac{dZ}{dx_i}$	$\frac{dC}{db} = \frac{dC}{dP} * \frac{dP}{dZ} * \frac{dZ}{db}$

$\frac{dC}{dw_i} = -2(Y - P) * \sigma'(Z) * x_i$	$\frac{dC}{dx_i} = -2(Y - P) * \sigma'(Z) * w_i$	$\frac{dC}{db} = -2(Y - P) * \sigma'(Z) * 1$

This is the math to compute how one weight, and one input need to be adjusted, but for a complete gradient this needs to be done for all weights and inputs at this neuron. To do this, only w and x need to change as they are the only values that are affected by changing the index on the summation. If repeated for all indexes, this would result in N input and weight derivatives and 1 bias derivatives. This can be described as a vector pointing in the direction that will cause the cost to increase the fastest, but as large costs are bad, this gradient can be multiplied by a negative one, switching the direction. This leaves a negative gradient vector that shows how one neuron should adjust its parameters to decrease the cost the most, shows how the next layer

needs to adjust its outputs. This needs to be done for every neuron in the output layer before backpropagating through to the output. This given enough time generally results in a neural network that fits a pattern and can predict outputs given new inputs. This is incredible, but the calculations shown do one weight, one input, and the bias for one layer, and it is not simple. When training large neural networks, this math might have to be done hundreds of thousands of times for every piece of data being used to train, which both takes significant time and energy.

This is a lot of calculations and doesn't necessarily result in fast learning of patterns. Models can take a long time to train, depending on how well the math allows the fitting of the patterns, meaning that ensuring proper math is important. While the basic NN is the most common, it is not always the best for every application, with the data being fit playing a critical role in training speed and the amount of training being conducted.

4.2.3 Training algorithm

The method of conducting these calculations can be improved to better use the resources of the computer. Matrix math is generally faster on computers as each element can be given to a different core, spreading the calculations and running training simultaneously. This can allow multiple computers to train the same model at the same time, and is why servers of GPUs are often used to train large models. GPUs have a lot of cores, allowing for a lot of parallel computation and with multiple GPUs, this can speed up training quite a bit. While the simplified training can be done in parallel, the matrices generally work better and are better optimized. In addition to running better on computers, the simplification carried out where the summation was ignored can not be used for some neuron equations, and while, if possible, the simple method builds a better understanding, it has its issues.

While the computer generally operates better, it also makes sense due to the quantity of parameters. Each neuron has multiple indices of summation, each with weights and inputs, and normally there are multiple neurons, resulting in many parameters needing to be adjusted. The

simplified math attempts to ignore this by only looking at one index at a time, but this is not possible for all neuron equations. The matrix method of training simply does all the neurons and neuron inputs at the same time and does not ignore the summation. While in the end it results in the same training, this more complicated method will only be used when absolutely necessary, as this section is exploring how math impacts starting models, making complicated methods not useful.

This section uses both methods to allow for connections to be drawn between the simple and complex training systems. These concessions will allow understanding of the basics of this matrix version of training, allowing for understanding in the latter sections. It starts off almost identically to the first step in the previously discussed method with a composite function seen below. The big difference is the subscript which denotes multiple outputs and inputs as they are calculated at the same time. One important note is the weight and bias with the added index k, this refers to the current neuron being examined. Each of these equations is for one neuron per k value, meaning each of these equations should be treated as vector outputs, as there are multiple k values. In the math, we will take these vectors and push them into a matrix to allow matrix multiplication better with more complex activation functions.

$$C_k = (Y - P_k)^2$$

$$P_k = \sigma(Z_k)$$

$$Z_k = \sum_{i=}^{N} W_{k,i}\ X_i + b_k$$

Where k is a neuron, N is a number of inputs, W is a weight matrix, X is an input vector, b is a vector(one bias per neuron), and σ is the activation function.

After defining the function, it's time to take derivatives. The derivatives of these equations are unusual as they are at least vectors as they input and output vectors. This allows us to handle the summation and similar repeated calculations as well. The proper term for what is being used is a Jacobian matrix, and while understanding this is not fully necessary, it can help. The basics are that each row of a Jacobian matrix is an output and each column is an input. This allows for any number of outputs and inputs per output, which is exactly what is needed for these equations.

$$\frac{dC_{k_0}}{dP_{k_1}} = \begin{bmatrix} \frac{dC_0}{dP_0} & \frac{dC_0}{dP_1} & \cdots & \frac{dC_0}{dP_{k_1}} \\ \frac{dC_1}{dP_0} & \frac{dC_1}{dP_1} & \cdots & \frac{dC_1}{dP_{k_1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dC_{k_0}}{dP_0} & \frac{dC_{k_0}}{dP_1} & \cdots & \frac{dC_{k_0}}{dP_{k_1}} \end{bmatrix}$$

This indexing is slightly confusing as the output index and input index are both k in the equation. Both k_0 and k_1 are the same, and being different does not make much sense. This is allowed in a sense as the derivatives when the two k indexes are different go to 0 as seen in the next box.

$$\frac{dC_{k_0}}{dP_{k_1}} = \begin{bmatrix} \frac{dC_0}{dP_0} & 0 & \cdots & 0 \\ 0 & \frac{dC_1}{dP_1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{dC_{k_0}}{dP_{k_1}} \end{bmatrix}$$

This makes more sense and written in this way, it can also be explained as a vector multiplied by an identity matrix to adjust the dimensions. As there are several outputs but only one input per output, it would make more sense to use a vector to contain the partial derivatives. If this is done, the dimensions for matrix multiplication would not work, but if multiplied by an identity, which does not change values, the dimensions can be changed to a square and results in the above matrix.

The diagonal of this matrix can be computed with:

$$\frac{dC_i}{dP_i} = -2\,(Y - P_i)$$

$$\frac{dP_{k_0}}{dZ_{k_1}} = \begin{bmatrix} \frac{dP_0}{dZ_0} & \frac{dP_0}{dZ_1} & \cdots & \frac{dP_0}{dZ_{k_1}} \\ \frac{dP_1}{dZ_0} & \frac{dP_1}{dZ_1} & \cdots & \frac{dP_1}{dZ_{k_1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dP_{k_0}}{dZ_0} & \frac{dP_{k_0}}{dZ_1} & \cdots & \frac{dP_{k_0}}{dZ_{k_1}} \end{bmatrix}$$

Normally the activation functions' derivative is just a diagonal matrix for the same reason, but some activation functions do not have the same input as output index, like softmax. This is caused by the fact that the softmax function results in an output that sums to 1, meaning each output is dependent on the other outputs.

$$\frac{dC_{k_0}}{dZ_{k_1}} = \frac{dC_{k_0}}{dP_{k_1}} \times \frac{dP_{k_0}}{dZ_{k_1}} = \begin{bmatrix} \frac{dC_0}{dZ_0} & \frac{dC_0}{dZ_1} & \cdots & \frac{dC_0}{dZ_{k_1}} \\ \frac{dC_1}{dZ_0} & \frac{dC_1}{dZ_1} & \cdots & \frac{dC_1}{dZ_{k_1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dC_{k_0}}{ZP_0} & \frac{dC_{k_0}}{dZ_1} & \cdots & \frac{dC_{k_0}}{dZ_{k_1}} \end{bmatrix}$$

Note the indexes do not appear to cancel, but do because k_0 and k_1 are really the same index.

$\frac{dZ_k}{dW_{k,i}} = \begin{bmatrix} \frac{dZ_0}{dW_{0,0}} & \frac{dZ_0}{dW_{0,1}} & \cdots & \frac{dZ_0}{dW_{0,i}} \\ \frac{dZ_1}{dW_{1,0}} & \frac{dZ_1}{dW_{1,1}} & \cdots & \frac{dZ_1}{dW_{1,i}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dZ_k}{dW_{k,0}} & \frac{dZ_k}{dW_{k,1}} & \cdots & \frac{dZ_k}{dW_{k,i}} \end{bmatrix}$	$\frac{dZ_k}{dX_i} = \begin{bmatrix} \frac{dZ_0}{dX_0} & \frac{dZ_0}{dX_1} & \cdots & \frac{dZ_0}{dX_i} \\ \frac{dZ_1}{dX_0} & \frac{dZ_1}{dX_1} & \cdots & \frac{dZ_1}{dX_i} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dZ_k}{dX_0} & \frac{dZ_k}{dX_1} & \cdots & \frac{dZ_k}{dX_i} \end{bmatrix}$	$\frac{dZ_{k_0}}{db_{k_1}} = \begin{bmatrix} \frac{dZ_0}{db_0} & \frac{dZ_0}{db_1} & \cdots & \frac{dZ_0}{db_{k_1}} \\ \frac{dZ_1}{db_0} & \frac{dZ_1}{db_1} & \cdots & \frac{dZ_1}{db_{k_1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dZ_{k_0}}{db_0} & \frac{dZ_{k_0}}{db_1} & \cdots & \frac{dZ_{k_0}}{db_{k_1}} \end{bmatrix}$
The coefficient of $W_{k,i}$ is X_i $\frac{dZ_k}{dW_{k,i}} = \begin{bmatrix} X_0 & X_1 & \cdots & X_i \\ X_0 & X_1 & \cdots & X_i \\ \vdots & \vdots & \ddots & \vdots \\ X_0 & X_1 & \cdots & X_i \end{bmatrix}$	The coefficient of X_i for output k is $W_{k,i}$ $\frac{dZ_k}{dX_i} = \begin{bmatrix} W_{0,0} & W_{0,1} & \cdots & W_{0,i} \\ W_{1,0} & W_{1,1} & \cdots & W_{1,i} \\ \vdots & \vdots & \ddots & \vdots \\ W_{k,0} & W_{k,1} & \cdots & W_{k,i} \end{bmatrix}$	$if\ (k_0 = k_1)\ then\ \frac{dZ_{k_0}}{db_{k_1}} = 1,$ $else\ \frac{dZ_{k_0}}{db_{k_1}} = 0$ $\frac{dZ_{k_0}}{db_{k_1}} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$ This is an identity matrix and does not change values of other matrices when multiplied.

$\frac{dC_k}{dW_{k,i}} = \frac{dC_{k_0}}{dP_{k_1}} \frac{dP_{k_0}}{dZ_{k_1}} \frac{dZ_k}{dW_{k,i}}$	$\frac{dC_k}{dX_i} = \frac{dC_{k_0}}{dP_{k_1}} \frac{dP_{k_0}}{dZ_{k_1}} \frac{dZ_k}{dX_i}$	$\frac{dC_{k_0}}{db_{k_1}} = \frac{dC_{k_0}}{dP_{k_1}} \frac{dP_{k_0}}{dZ_{k_1}} \frac{dZ_{k_0}}{db_{k_1}}$

When multiplying these partial derivatives, k_0 and k_1 should be treated identically.

$\frac{dC_k}{dW_{k,i}} = \frac{dC_{k_0}}{dP_{k_1}} \times \frac{dP_{k_0}}{dZ_{k_1}} \times \frac{dZ_k}{dW_{k,i}} = \begin{bmatrix} \frac{dC_0}{dW_{0,0}} & \frac{dC_0}{dW_{0,1}} & \cdots & \frac{dC_0}{dW_{0,i}} \\ \frac{dC_1}{dW_{1,0}} & \frac{dC_1}{dW_{1,1}} & \cdots & \frac{dC_1}{dW_{1,i}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dC_k}{dW_{k,0}} & \frac{dC_k}{dW_{k,1}} & \cdots & \frac{dC_k}{dW_{k,i}} \end{bmatrix}$	$\frac{dC_k}{dX_i} = \frac{dC_{k_0}}{dP_{k_1}} \times \frac{dP_{k_0}}{dZ_{k_1}} \times \frac{dZ_k}{dX_i} = \begin{bmatrix} \frac{dC_0}{dX_0} & \frac{dC_0}{dX_1} & \cdots & \frac{dC_0}{dX_i} \\ \frac{dC_1}{dX_0} & \frac{dC_1}{dX_1} & \cdots & \frac{dC_1}{dX_i} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dC_k}{dX_0} & \frac{dC_k}{dX_1} & \cdots & \frac{dC_k}{dX_i} \end{bmatrix}$	$\frac{dC_{k_0}}{db_{k_1}} = \frac{dC_{k_0}}{dP_{k_1}} \times \frac{dP_{k_0}}{dZ_{k_1}} \times \frac{dZ_{k_0}}{db_{k_1}} = \begin{bmatrix} \frac{dC_0}{dZ_0} & \frac{dC_0}{dZ_1} & \cdots & \frac{dC_0}{dZ_{k_1}} \\ \frac{dC_1}{dZ_0} & \frac{dC_1}{dZ_1} & \cdots & \frac{dC_1}{dZ_{k_1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dC_{k_0}}{ZP_0} & \frac{dC_{k_0}}{dZ_1} & \cdots & \frac{dC_{k_0}}{dZ_{k_1}} \end{bmatrix}$ As dz/db is the identity, thus this is just dC/dZ

You might note that the adjustment matrices are too large in one dimension to adjust their respective parameters. This is caused by multiple adjustments needing to be made or the shape of the matrix having to work with other matrices. For example, DC/dX, which is the change being requested of the next layer, is affected by each neuron as each neuron takes all the inputs. This results in a matrix that is overly large, accounting for each of the adjustments wanted by each neuron. But for all of these misshapen matrices, the solution to the issue is the same and is just the summing of columns. This results in a vector of the length of the number of columns and is exactly the right size, and for the DC/dX matrix uses all the desired adjustments by simply adding, ensuring proper changes are made.

4.2.4 Summary

The math of the operation and training of these models sheds light on how these models perform. These models are the standard and go-to models for good reason, balancing training speed and the ability to fit many different types of functions. The two main characteristics needing examination are operation and training, as well as what situations this model is likely to perform well and not well in.

This model runs quite fast during operation with the activation function being the slowest aspect of computation. This model is often used for regression and classification tasks, and while large inputs will slow it down, it is generally fast. The activation function runs once for each neuron, which limits its use, decreasing the often more complex calculations involved. This leaves the bulk of the computation and time running the model with the weighted sum, which combines the inputs. This is fast with matrix multiplication and is truly just adding and multiplying which does not take much time. This model is likely to work best during operation with more inputs than outputs, as this limits the activation function use, simplifying the math. While it is fast and operates smoothly, this model is known for slow training.

During training, the basic NN model is slow but is consistent and easy to utilize. Section 4.1 discusses the issues with activation function derivatives and small values resulting in slow training. This affects the basic NN model greatly, making the adjustment matrix generally small, making each improvement minimal. This is not always bad as it helps to mitigate adjustments in the wrong direction, but does make this model slow. Compared to models that will be discussed later, the basic NN trains relatively quickly with limited use of the activation function and its derivative. This model is also likely to be the best when it comes to fitting linear data, as while the activation function makes it nonlinear, the rest of the math is very linear. This all comes together to make a slow but reliable training model and a good model for comparison for the more experimental models discussed in the next sections. While there is not much that can be said about what situations this model will be good at seeing as no other models have been proposed, it is the best for its reliability and the amount of refinement in terms of computer integration. In a situation where nothing is known about the data, the basic NN is likely a good starting model pick.

4.3 SumLastNN

This section will go over a modification of the neuron equation that adds additional biases. This section uses information from a paper written by Carlow Metta, Gianluca Amato, and several others who designed and tested this modified neuron equation. This section will go over the modifications, reasons for the modifications, and use cases for the modified neuron (Metta, Carlo, et al). This neuron will be called the SumLast neuron due to the summation being the last step, this is not an official name.

4.3.1 SumLast Neuron Equation

The neuron equation for the SumLast model is a variation of the basic NN model equation that is rearranged to allow for more biases. In the basic neuron equation seen in section 3.2.1, there is one bias per input, and adding additional biases would make no sense as they would just be added together. This can also be seen by looking at the training math in either section 3.2.2 or 3.2.3, where the partial derivative of cost in terms of the biases is 1, meaning that it would be the same adjustment if you added multiple biases. But there can be more than one bias per neuron if the equation is rearranged.

SumLast Neuron Equation:

$$\sum_{i=0}^{N} w_i \sigma(x_i + b_i)$$

Where σ is the activation function, W_i is the weight for input i, x_i is input i, b_i is the bias for input i, and N is the number of inputs (neurons in previous layer).

This function has N inputs per neuron, meaning there are just as many biases as weights in this equation, and per neuron with this modification. This is possible because the activation

function is moved inside of the summation, allowing each bias to act as a unique horizontal shift for the N activation function. These added biases give more adjustability at each of these neurons with the cost of taking more computation during training and when predicting.

This increase in computation is caused by each neuron running a summation per input, which is N times as many as the basic NN model. This increase in activation function use allows for more biases as they are no longer just summed, but depending on the activation function, it can slow the model's operation significantly. Simple activation functions like ReLU can allow this model to operate much faster, but also make the multiple biases less impactful as they are basically just summed once, after weight multiplication, again. This makes the use case for this model much smaller than the basic NN, but it does have benefits.

Beyond just having more biases, the moving of the activation function also changes the possible outputs of this function. Most activation functions scale values between 0 and 1, which is not always desirable. The weights being multiplied by the activation function, and multiple activation functions being summed, results in not much of a range bounding with this neuron equation. This means that the output of this neuron has many more options, which is optimal for regression tasks, where values might get much larger. While generally unimportant due to output scaling, this adds value, and with the additional biases, should make this model great at finding complex patterns with larger outputs.

4.3.2 Training

This model requires training through the more complex variation due to the summation being moved. The equation components of the neuron equation with the cost function are given below. This will be differentiated to find the slope of cost in terms of all the weights and biases

as done previously. Refer to section 4.2.3 for explanations of the math and section 3.5 for conceptual understanding.

$$C_k = \left(Y - \sum_{i=1}^{N} W_{k,i}\ P_{k,i}\right)^2$$

$$P_{k,i} = \sigma\left(Z_{k,i}\right)$$

$$Z_{k,i} = X_i + b_{k,i}$$

Derivative for cost function:

$$\frac{dC_k}{dP_{k,i}} = \begin{bmatrix} \frac{dC_0}{dP_{0,0}} & \frac{dC_0}{dP_{0,1}} & \cdots & \frac{dC_0}{dP_{0,i}} \\ \frac{dC_1}{dP_{1,0}} & \frac{dC_1}{dP_{1,1}} & \cdots & \frac{dC_1}{dP_{1,i}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dC_k}{dP_{k,0}} & \frac{dC_k}{dP_{k,1}} & \cdots & \frac{dC_k}{dP_{k,i}} \end{bmatrix}$$

$$\frac{dC_k}{dP_{k,i}} = -2\left(Y - \sum_{i=0}^{N} W_{k,i}\,P_{k,i}\right)\begin{bmatrix} W_{0,0} & W_{0,1} & \cdots & W_{0,i} \\ W_{1,0} & W_{1,1} & \cdots & W_{1,i} \\ \vdots & \vdots & \ddots & \vdots \\ W_{k,0} & W_{k,1} & \cdots & W_{k,i} \end{bmatrix}$$

Derivative for activation function (often just a diagonal matrix):

$$\frac{dP_{k,i}}{dZ_{k,i}} = \begin{bmatrix} \frac{dP_{0,0}}{dZ_{0,0}} & \frac{dP_{0,1}}{dZ_{0,1}} & \cdots & \frac{dP_{0,i}}{dZ_{0,i}} \\ \frac{dP_{1,0}}{dZ_{1,0}} & \frac{dP_{1,1}}{dZ_{1,1}} & \cdots & \frac{dP_{1,i}}{dZ_{1,i}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dP_{k,0}}{dZ_{k,0}} & \frac{dP_{k,1}}{dZ_{k,1}} & \cdots & \frac{dP_{k,i}}{dZ_{k,i}} \end{bmatrix}$$

Derivative for Input and Bias:

It should be noted that they are identical and do not need to be computed during training.

$\frac{dZ_{k,i}}{dX_i} = \begin{bmatrix} \frac{dZ_{0,0}}{dX_0} & \frac{dZ_{0,1}}{dX_1} & \cdots & \frac{dZ_{0,i}}{dX_i} \\ \frac{dZ_{1,0}}{dX_0} & \frac{dZ_{1,1}}{dX_1} & \cdots & \frac{dZ_{1,i}}{dX_i} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dZ_{k,0}}{dX_0} & \frac{dZ_{k,1}}{dX_1} & \cdots & \frac{dZ_{k,i}}{dX_i} \end{bmatrix}$	$\frac{dZ_{k,i}}{db_{k,i}} = \begin{bmatrix} \frac{dZ_{0,0}}{db_{0,0}} & \frac{dZ_{0,1}}{db_{0,1}} & \cdots & \frac{dZ_{0,i}}{db_{0,i}} \\ \frac{dZ_{1,0}}{db_{1,0}} & \frac{dZ_{1,1}}{db_{1,1}} & \cdots & \frac{dZ_{1,i}}{db_{1,i}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dZ_{k,0}}{db_{k,0}} & \frac{dZ_{k,1}}{db_{k,1}} & \cdots & \frac{dZ_{k,i}}{db_{k,i}} \end{bmatrix}$
$\frac{dZ_{k,i}}{dX_i} = \begin{bmatrix} 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \ldots & 1 \end{bmatrix}$	$\frac{dZ_{k,i}}{db_{k,i}} = \begin{bmatrix} 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \ldots & 1 \end{bmatrix}$

$$\frac{dc}{dx} = \frac{dc}{dp} * \frac{dp}{dz} * \frac{dz}{dx}$$

$$\frac{dc}{db} = \frac{dc}{dp} * \frac{dp}{dz} * \frac{dz}{db}$$

The partial derivatives for weights are calculated separately and are below. They are separate as they are not a part of the equation Z.

$$\frac{dC_k}{dW_{k,i}} = \begin{bmatrix} \frac{dC_0}{dW_{0,0}} & \frac{dC_0}{dW_{0,1}} & \cdots & \frac{dC_0}{dW_{0,i}} \\ \frac{dC_1}{dW_{1,0}} & \frac{dC_1}{dW_{1,1}} & \cdots & \frac{dC_1}{dW_{1,i}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dC_k}{dW_{k,0}} & \frac{dC_k}{dW_{k,1}} & \cdots & \frac{dC_k}{dW_{k,i}} \end{bmatrix}$$

$$\frac{dC_k}{dW_{k,i}} = \begin{bmatrix} P_{0,0} & P_{0,1} & \cdots & P_{0,i} \\ P_{1,0} & P_{1,1} & \cdots & P_{1,i} \\ \vdots & \vdots & \ddots & \vdots \\ P_{k,0} & P_{k,1} & \cdots & P_{k,i} \end{bmatrix}$$

4.3.3 Summary

This model's operation is generally slower due to the extra activation functions being used and the potential for larger outputs. The increase to computation caused by the activation function is not much when working with mostly linear activation functions like ReLU and with models that keep the number of inputs and neurons small. While the computational cost can be mitigated, the arrangement of the equations does not limit the output, potentially resulting in

large values being cycled through the model. This can also be mitigated using a mix of the SumLast NN and Basic NN models to keep the values running through the network small. This model can run close to as fast as the basic NN if limiting activation function choices.

While this model can be made to run quickly, the more important aspect of this model is the additional weight and biases. These allow more patterns to be found with more adjustments, and this can make up for the speed depending on the situation. Some situations would benefit more from better pattern recognition than running faster speeds, like most medical identification applications. This all comes together to make this model likely to be a good choice in situations where operation speed does not matter so much as correct prediction of complicated patterns.

While this model has uses after training, the training of this model is computationally difficult. While several simplifications minimize computation, such as duplicate partial derivatives, the derivatives of the activation function take time to compute. The quantity of activation function derivatives and the possibility for larger outputs result in slow training with potential gains caused by the extra biases. While the SumLast model is not good when it comes to training speed, the potential for this model is large.

The SumLast model is optimal for situations where training speed is not important, nor when fast computation is needed, but it can be helpful for complex tasks. This model will likely perform well in situations that use its ability to have larger output with higher complexity when training speed is not important or the models are small enough. Large models using this system will train much slower, but small models will train fast enough not to be noticeable. Depending on the situation, this model could prove to be extremely beneficial, yet it is clear that it will only work in certain situations.

4.4 WNN

This section discusses a variant of the SumLast model attempting to partially limit the range and stretch the domain while maintaining the additional biases (Metta, Carlo, et al). This model is based on the SumLast model with adjustments to follow more in line with the basic NN in an attempt to improve pattern recognition by moving the weight inside the activation function. This model was also designed around not starting with random weight and bias, instead using probability-based weight and bias initialization, but this has not been finalized (Nguyen and Widrow). As such, this model is just part of an experimental model and will likely struggle during training without the starting advantages.

This is an experimental model taking inspiration from the article about the SumLast model and Nguyen and Widrow's paper on starting weights and biases. This is an untested model and is included in this paper to explore the idea of situationally generated neuron equations designed to be even more situational.

4.4.1 The Neuron Equation

This follows an almost identical equation to the SumLastNN with the one difference being the weight. Having the weight inside the activation function, allows for the activation function to be stretched or squashed. For example, with the sigmoid activation function, the domain between -4 and 4 has lots of change with everything outside being either 0 or 1 as the output. Moving the weight inside the activation function allows for this to be adjusted, making the activation function more adjustable. The activation function is the only source of nonlinearity, so instead of changing the amplitude as with the SumLast model, this model changes the domain. This allows the model to have greater contral over nonlinearity during training while maintaining the additional weights.

Neuron equation:

$$\sum_{i=0}^{N} \sigma(w_i x_i + b_i)$$

Where σ is the activation function, w_i is the weight for input i, x_i is input i, b_i is the bias for input i, and N is the number of inputs (neurons in previous layer).

This model is similar to the SumLast model in terms of operation and operational challenges, which can be read in section 4.3. The only change during operation is that the weight is not multiplied by the activation function, resulting in the activation function controlling the output range to a greater extent, generally keeping values smaller and speeding up computation.

4.4.2 Training

The training for this model is similar to what is demonstrated in 4.3.3, with the difference being that the weight matrix is calculated like the bias and input of the SumLast model.

$$C_k = \left(Y - \sum_{i=1}^{N} P_{k,i}\right)^2$$

$$P_{k,i} = \sigma\left(Z_{k,i}\right)$$

$$Z_{k,i} = W_{k,i} X_i + b_{k,i}$$

Derivative for loss function:

$$\frac{dC_k}{dP_{k,i}} = \begin{bmatrix} \frac{dC_0}{dP_{0,0}} & \frac{dC_0}{dP_{0,1}} & \cdots & \frac{dC_0}{dP_{0,i}} \\ \frac{dC_1}{dP_{1,0}} & \frac{dC_1}{dP_{1,1}} & \cdots & \frac{dC_1}{dP_{1,i}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dC_k}{dP_{k,0}} & \frac{dC_k}{dP_{k,1}} & \cdots & \frac{dC_k}{dP_{k,i}} \end{bmatrix}$$

$$\frac{dC_k}{dP_{k,i}} = -2\left(Y - \sum_{i=0}^{N} P_{k,i}\right)\begin{bmatrix} 1_{0,0} & 1_{0,1} & \cdots & 1_{0,i} \\ 1_{1,0} & 1_{1,1} & \cdots & 1_{1,i} \\ \vdots & \vdots & \ddots & \vdots \\ 1_{k,0} & 1_{k,1} & \cdots & 1_{k,i} \end{bmatrix}$$

The indexes given on the 1s are for when the multiplication occurs between the matrix and the cost derivative as it depends on the indexes.

Derivative for activation function:

$$\frac{dP_{k,i}}{dZ_{k,i}} = \begin{bmatrix} \frac{dP_{0,0}}{dZ_{0,0}} & \frac{dP_{0,1}}{dZ_{0,1}} & \cdots & \frac{dP_{0,i}}{dZ_{0,i}} \\ \frac{dP_{1,0}}{dZ_{1,0}} & \frac{dP_{1,1}}{dZ_{1,1}} & \cdots & \frac{dP_{1,i}}{dZ_{1,i}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dP_{k,0}}{dZ_{k,0}} & \frac{dP_{k,1}}{dZ_{k,1}} & \cdots & \frac{dP_{k,i}}{dZ_{k,i}} \end{bmatrix}$$

Derivative for Input and Bias:

$\frac{dZ_{k,i}}{dX_i} = \begin{bmatrix} \frac{dZ_{0,0}}{dX_0} & \frac{dZ_{0,1}}{dX_1} & \cdots & \frac{dZ_{0,i}}{dX_i} \\ \frac{dZ_{1,0}}{dX_0} & \frac{dZ_{1,1}}{dX_1} & \cdots & \frac{dZ_{1,i}}{dX_i} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dZ_{k,0}}{dX_0} & \frac{dZ_{k,1}}{dX_1} & \cdots & \frac{dZ_{k,i}}{dX_i} \end{bmatrix}$	$\frac{dZ_{k,i}}{dW_{k,i}} = \begin{bmatrix} \frac{dZ_{0,0}}{dW_{0,0}} & \frac{dZ_{0,1}}{dW_{0,1}} & \cdots & \frac{dZ_{0,i}}{dW_{0,i}} \\ \frac{dZ_{1,0}}{dW_{1,0}} & \frac{dZ_{1,1}}{dW_{1,1}} & \cdots & \frac{dZ_{1,i}}{dW_{1,i}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dZ_{k,0}}{dW_{k,0}} & \frac{dZ_{k,1}}{dW_{k,1}} & \cdots & \frac{dZ_{k,i}}{dW_{k,i}} \end{bmatrix}$	$\frac{dZ_{k,i}}{db_{k,i}} = \begin{bmatrix} \frac{dZ_{0,0}}{db_{0,0}} & \frac{dZ_{0,1}}{db_{0,1}} & \cdots & \frac{dZ_{0,i}}{db_{0,i}} \\ \frac{dZ_{1,0}}{db_{1,0}} & \frac{dZ_{1,1}}{db_{1,1}} & \cdots & \frac{dZ_{1,i}}{db_{1,i}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{dZ_{k,0}}{db_{k,0}} & \frac{dZ_{k,1}}{db_{k,1}} & \cdots & \frac{dZ_{k,i}}{db_{k,i}} \end{bmatrix}$
$\frac{dZ_{k,i}}{dX_i} = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,i} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,i} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,i} \end{bmatrix}$	$\frac{dZ_{k,i}}{dW_{k,i}} = \begin{bmatrix} x_0 & x_1 & \cdots & x_i \\ x_0 & x_1 & \cdots & x_i \\ \vdots & \vdots & \ddots & \vdots \\ x_0 & x_1 & \cdots & x_i \end{bmatrix}$	$\frac{dZ_{k,i}}{db_{k,i}} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}$

$$\frac{dc}{dx} = \frac{dc}{dp} * \frac{dp}{dz} * \frac{dz}{dx}$$

$$\frac{dc}{db} = \frac{dc}{dp} * \frac{dp}{dz} * \frac{dz}{db}$$

4.4.3 Summary

This model should perform very similarly to the SumLast model in operation. The one difference is that the WNN model's outputs are likely to be smaller due to the weight being inside the activation function. The range of this model is restricted to the range of the activation function times the number of inputs. This is a restriction that has its downsides but will also allow this model to interact with other models much more easily, as its outputs will not be as large. This will also improve the overall speed of operation as small values are easier to

calculate. This means that during operation, this model is likely to be faster than the SumLast model but slower than the basic NN.

During training, however, this model will likely take more time than the sumLast model, though it is hard to tell. There is more multiplication required during training in the last step due to the weight being a coefficient of the input. This will likely decrease the performance of the model during training compared to the SumLast model, but it is hard to tell. Both of these models are very similar and are likely to perform well in similar situations. However, this model will likely operate better for tasks where the output size is more bounded than the SumLast model.

4.5 Basic Convolution Neural Network

The Convolution Neural Network, also known as a CNN, is an extremely common type of machine learning using the basic neuron equation discussed in section 4.2.1. Unlike the basic NN, this model does not use multiple neurons and instead uses one neuron multiple times. This model segments the input into subsets, running each subset through the same neuron resulting in an output of a size comparable to the number of segments. This makes this model small, even with a lot of inputs, as it has only one neuron operating on small subsets of data, limiting a number of weights and biases. This sharing of weights and biases results in this model being used for feature extraction, which is where the model identifies patterns in the subsets instead of the entire data. This is useful in images as often subjects are not located in the same location, making this type of subset style model highly effective.

This section covers the operation and training of the basic conclusion style network. The math presented in this section comes from an article written by Gavneet Singh Chadha and

Andreas Schwung about a variant of the basic CNN. The next section goes further into that article research, but a foundation is needed before exploring the more sophisticated variation.

4.5.1 Neuron Equation & Operation

The basic convolution style layer is identical mathematically to the basic NN style layer, just with a different data flow. This type of layer is often followed by a basic NN layer as there is not much possible training to this layer due to only having one neuron. Unlike all previous layer types discussed this layer has a single neuron applied multiple times. The neuron equation it uses is the same as the basic NN layer.

The basic Convolution Neuron Equation:

$$\sigma(\sum_{i=0}^{N} w_i x_i + b)$$

This equation is identical to the previously discussed NN layer with a weighted sum put through an activation function. In a basic NN, each neuron takes all the inputs and sums them to one output, each neuron has individual weights and biases. The CNN takes a subset of the input data, meaning a small section of the input to pass through its neuron. This is often explained as a magnifying glass slowly moving across an image, where you can only see a small portion of the image at a time. For the basic NN, this magnifying glass, more properly called the kernel, moves one data point at a time, going across all the data with each subsection of data getting its own output. This results in outputs smaller than the input, depending on the size of the kernel, with large kernels meaning more inputs at once to a neuron, but a smaller number of outputs. If the kernel is the magnifying glass, this can be thought of as having less room to move the

magnifying glass if it is large. This type of model is designed with multiple dementia inputs like images, where the position of the pixel patterns.

The kernel system allows for focused identification of specific patterns like heads or ears no matter the location. As the kernel moves, it examines each part of the image separately, allowing for what is called feature extraction. This isolates specific characteristics that are important to classification, like different ear shapes can be used to identify different animal varieties. This also decreases the size of the layer as it only stores one set of weights based on the kernel size and one bias.

While this model is small in size, it still takes a lot of computational power. While there is only one neuron, it is used many times, calling on the activation function each time. It runs faster than basic NN when there are large amounts of inputs and outputs, as it only takes a part of the input to give an output. But there are cases where the basic NN runs faster, especially when the input is small.

4.5.2 Training of CNN

The training is nearly identical to the basic NN, see section 4.2.2. The difference is that this computation must be computed for each kernel. As each kernel is basically a separate input, they must all be computed to properly update the weights and biases based on the entire image, not just one section. In other words, it trains the one neuron for every subsection of the imputed data.

This makes the training faster than the basic NN in the same situation, it will operate faster. This means that this model is a good pick for large inputs with many outputs, as the kernel system allows this to operate quickly. The increase in speed during training is less significant than operation, but still substantial.

4.5.3 Summary

This layer is extremely small compared with its shared weights and biases. During operation, it is faster computationally compared to a basic NN with the same number of outputs as it uses a subset of inputs minimizing computation per output. While this model is faster in most situations, it only has one set of weights and bias, limiting the patterns that can be identified. This makes this model better at highlighting potential important features of data and not making predictions. This means that this model, during operation, is best in situations where there is a multi-layer setup, where this style of neuron equation can be used to preprocess large inputs. This is a pretty specific scenario, but fairly common with examples including speech, and image recognition. In addition, it is small, making it great for situations where storage is limited and models must be small.

Training of a CNN model is potentially faster than that of a basic NN. There are benefits to this model with large inputs and this holds for training. While during training it operates even more similarly to a basic NN with the same number of outputs, it is faster due to smaller input sizes. While the CNN is faster, it is not much faster, and as it has much less ability to train patterns, the training it does do is less beneficial. This model has its place and while it operates fast, it is normally chosen for its ability to handle large inputs quickly.

4.6 Exponential Convolution Neural Network

The Exponential Convolution Neural Network follows the same data input structure as the basic CNN with just a more complex neuron equation. The more complex neuron equation results in slightly more weights and an ability to understand more complex patterns. Most types of neural network layers have an untrainable nonlinearity to them, but this function with its

trainable exponential weight could result in significant improvements. The information from this section is from Gavneet Singh Chadha and Andreas Schwung, who wrote a paper on this method of changing the convolution neural network. Unfortunately, they have not yet published results, so all conclusions about improvements in the section are based on how the math operates.

4.6.1 Neuron Equation

The only difference between the exponential CNN and the basic CNN as discussed in section 4.5 is the neuron equation. Unlike the basic CNN which uses the same equation as the basic NN, this model has two separate weights per input in addition to the standard basis. This second weight is an exponent that is applied to the input, which can take some more computational work. Below is the neuron equation.

The Exponential Neuron Equation:

$$\sigma(\sum_{i=0}^{N} w_{0,i} x_i^{w_{1,i}} + b)$$

The exponential neuron equation is very similar to the equation used by the standard CNN model with the exception of the additional weight matrix w_1. The variable w_0 refers to the standard multiplicative weight. The operation of this type of equation does not take much additional time, but can cause some issues when training, as discussed in section 4.6.2. This equation should be much better at fitting and identifying nonlinear functions, especially exponential relationships. Unlike previously discussed layers, which only train the linear parts of the function, leaving all nonlinear fitting ability to the fixed activation function. This exponential convolution adds the ability to train non-linearity through the exponential weight, and while this type of nonlinearity that is being trained is relatively common in real-world data, the method

discussed also applies a more standard untrainable activation function. This results in a model that has more nonlinearity and this nonlinearity is partially trainable.

While this model should be better at identifying patterns than the basic CNN, it still only has one neuron. While it might improve how well it can distinguish features, it will struggle to make predictions without a more standard type of model assisting it.

4.6.2 Exponential CNN training.

For the training of this neuron equation, there are some differences compared to the basic CNN caused by the exponential term. This model will still use backpropagation and mean squared error as its loss function. It should be noted that there are some values that do not work as inputs to this function due to the exponent, which will be discussed at the end of this section

Separate the function into its component parts, including the loss function

$$C = (Y - P)^2$$

$$P = \sigma(z)$$

$$z = \sum_{i=0}^{N} w_{0,i} x_i^{w_{1,i}} + b$$

Then, take the partial derivative for one input inside of one kernel.

$$\frac{dC}{dP} = -2(Y - P)$$

$$\frac{dP}{dz} = \sigma'(z)$$

$$\frac{dz}{dw_{0,i}} = x_i^{w_{1,i}}, \frac{dz}{dw_{1,i}} = w_{0,i} ln(x_i) x_i^{w_{1,i}}, \frac{dz}{db} = 1, \frac{dz}{dx_i} = w_{0,i} w_{1,i} x_i^{w_{1,i}-1}$$

These derivatives show the best possible way to change the parameters but also some limitations of this model during training. The natural log function has a domain 0<x<infinity, where zero gives -infinity. In our partial derivative, we take a natural log of our input x_i, which generally has a range of $0 \leq x \leq 1$, as these values are small and fast to compute. Unfortunately, if we try to run this through the partial derivative, we will get an infinity when x is 0, which will result in failed training. Unusual oddities like this are sometimes hard to keep track of during the building process of a mode structure but can easily cause training failures or, worse, partial success.

The reason this is so important is that the data input is not the only thing scaled from 0 to 1. Most activation functions also have zero as a possible output, making it hard to put this exponential layer after most other layers. While the computing may not be much more complex, the use cases of this layer are generally limited to the first layer, with data scaled always to be larger than one, even if only by a small amount. One other note of importance about scaling is that the smallest values of the natural log are near one, not zero, making scaling around one result in smaller numbers and, thus, generally faster computations.

4.6.3 Use Cases/ Summary

This model is very similar to the basic CNN discussed in section 3.6 and generally can be used in the same ways. The basic CNN is good with large inputs as it reuses the same weights and biases, meaning minimal size, and each calculator is smaller. It also has the downside of having to run calculations more times, which can result in longer run times depending on the input size. Generally speaking, this type of layer is good for the start of models with large inputs to help identify features.

The exponential CNN has the same data flow and as a result similar training to the basic CNN. The additional weight, being an exponent, has several implications both in training and in operation. This exponent slows down the calculator as there are more steps to compute than in the basic CNN, and while there is possibly a minimal increase per calculation, there are a lot of calculations potentially adding up to a significant rise in calculation complexity. The second difference is the type of patterns this model can identify, which is the reason why this model might be a good choice depending on the situation. As there is an additional weight, there are more things being adjusted and thus more possible curves this one layer can represent. This makes this a great choice of model when the basic CNN is not identifying patterns and more complexity is needed without increasing model size.

4.7 Vis

This layer is completely different from any other model discussed. This was created to resolve the issues with non-visual learning as well as to make AI easier to explain. This layer runs on a different paradigm from any other model, for a more visual operation that is understandable. Most AI models and structures, including the previously discussed model, are extremely hard to visualize due to extreme dimensionality. Take a very simple NN single layer with three inputs and two outputs. This layer would take five dimensions to visualize and would likely make little to no sense. We as humans can generally understand function up to the third dimension, and with effort to the fourth, but the fifth dimension is tough to visualize and even harder to understand. While five dimensions are hard to understand, most models have thousands

of inputs and outputs, making it basically pointless to try. This visual model can not easily show training, but its output is extremely visual and operates in two dimensions, no matter the number of outputs or inputs, making it relatively easy to understand.

This model is not based on any sources and is purely here to show that the structures used in the field aren't necessarily the only way. This model was designed to show how a neural network model operates without high-dimensional graphs. This might not train well, but in situations where understanding a model is more important, such as when teaching about AI, this model is a clear choice.

4.7.1 Operation

This model was designed to output a function that points to the desired output. This means that if the output is plotted, it will be a function that points to the desired answer like a hand might. This can be thought of as a spinning arrow that points to the category that is desired. This model has not had much refinement or consideration for training, most likely resulting in poor results in the testing section.

Unlike the previously discussed layers, which are functions with numerical input and output, this model output takes in a rational number and outputs a function. This function is two-dimensional and built of a sine function and is thus able to be visualized. To get numerical outputs to allow for data fitting, the function is integrated. For multiple outputs the domain is just segmented to allow for multiple intrals.

To get a function that points this model was built on sines and operates on a similar principle to Fourier analysis, and its ability to match any function. While it is not using a Fourier transform, it does use a sum of many sign functions to build its output function.

The output function is defined as:

$$\sum_{i=0}^{N} sin(w_i(x_i + b_i)t)$$

The output value is defined as:

$$\int_{a_l}^{b_l} \sum_{i=0}^{N} sin(w_i(x_i + b_i)t)dt$$

Where N is the number of inputs, and a and b are the ranges where l is the specific output. W is the weight, x is the input, b is the bias, all dependent on the current input shown by the index.

This model does not use neurons, which has pros and cons. For all the previous models to add more weights and biases to identify more complex patterns, all that is needed is more neurons. For this function, it is based on the number of inputs, so to increase the complexity of the patterns, more inputs are needed. This allows this model to be visible, and simply duplicating the inputs does allow for more pattern recognition ability. This means that the more inputs, the more computational complexity, but that having lots of outputs does not. This results in the first model that operates faster when there are fewer inputs and lots of outputs. While there aren't many situations like this, in this situation it could perform extremely quickly.

One more important note about the neuron equation is that it does not have an adjustable activation function. While an adjustable activation function might improve this model, it has not been tested, meaning this model is stuck with a sine function. While the sine function is flexible and useful, as shown by Fourier and how it can fit any function, the proper activation function significantly speeds up training for most models.

4.7.2 Training

The training of this model still uses backpropagation with gradient descent and thus can operate along with the rest of the described models despite its lack of neurons. The integral can be moved inside the summation and applied, making the derivatives long but not requiring any matrices.

Starting equation:

$$C_l = (Y - P_l)^2$$

$$P_l = \int_{a_l}^{b_l} \sum_{i=0}^{N} sin(w_i(x_i + b_i)t)dt$$

Taking integral:

$$P_l = \sum_{i=0}^{N} \frac{cos(w_i(x_i+b_i)a_l) - cos(w_i(x_i+b_i)b_l)}{w_i(x_i+b_i)}$$

Taking derivatives:

$A=(x_i + b_i)$
$\frac{dC_l}{dP_l} = -2(Y - P_l)$
$\frac{dP_l}{dw_i} = \sum_{i=0}^{N} \frac{w_i A(sin(w_i Ab_l)b_l - sin(w_i Aa_l)a_l) - cos(w_i Aa_l) + cos(w_i Ab_l)}{w_i^2 A}$
$\frac{dP_l}{dx_i} = \sum_{i=0}^{N} \frac{w_i A(sin(w_i Ab_l)w_i b_l - sin(w_i Aa_l)a_l w_i) - w_i(cos(w_i Aa_l) - cos(w_i Ab_l))}{(w_i A)^2}$
$\frac{dP_l}{da_l} = -sin(w_i(x_i + b_i)t_0)$

$\frac{dP_l}{db_l} = sin(w_i(x_i + b_i)t_l)$

$\frac{dC_l}{dw_i} = \frac{dC_l}{dP_l}\frac{dP_l}{dw_i}$
$\frac{dC_l}{dx_i} = \frac{dC_l}{dP_l}\frac{dP_l}{dx_i}$
$\frac{dC_l}{da_l} = \frac{dC_l}{dP_l}\frac{dP_l}{da_l}$
$\frac{dC_l}{db_l} = \frac{dC_l}{dP_l}\frac{dP_l}{db_l}$

These derivatives are simple compared to sum models, but still have to be applied a similar amount of times as other models. While it may appear simpler, it still takes time to compute. There are quite a few sine and cosine functions used during the training process, which can really slow down training. This model has much room for improvement, but does train and allows for a more visual explanation of complex machine learning.

4.7.3 Summary

During ordinary operation, this model is computationally fast with all the computation dependent on the number of inputs. Unlike most models, the calculator amount is based on input size instead of output size. The activation function in the BasicNN, like most of the other models is run at each neuron, as with this model, the sine function is run at each input. This means that the best way to better fit the data is increasing the number of inputs by duplicating the input or

starting with a larger input count. To decrease computation, all that is needed is to decrease the input size, but this is not always possible.

This model will not be the fastest model, but it should be relatively comparable to most models in computation speed. Unfortunately, this is not the case for training due to the additional sine and cosine functions during training caused by the integral. This will make this model slower than something like the BasicNN model depending on the input and output size of the situation.

This model, while untested, does have situations where it operates better than any other. While most models focus on accuracy, this model focuses on understandability. This makes it the best choice for teaching about the operation of machine learning at a conceptual level. It is also likely to perform well on data with a sine-based pattern. It is important to keep in mind that while machine learning operation is important, understanding the operation is also important.

6 Part V: Experiment 1 Methods and Results

Experiment one is testing the speed of a neural network on regression tasks. Regression tasks are predictions of output on a function and are the same thing as curve fitting. In this case, we will test several models discussed below on three linear functions and three nonlinear functions. Each test will compare different aspects of a model, and in the results, we will connect the performance with the math that is stated above. While simple, this test will show that there is no one best model and that it depends on the data it is being trained on, while giving some reasons why varying models perform well, which can be used to improve future training.

6.1 Models:

Training rate has not been mentioned before in detail as it does not affect the math. It is simply a constant that is multiplied to the gradient. The training rate is normally less than one, meaning that the gradient and thus the change to the weights and biases is decreased. Smaller adjustments often mean it takes the model longer to learn, but that they can achieve higher accuracy at the end of training. In this experiment, most models are using a training rate of 1 with a few exceptions that require lower training rates to operate.

The below tables give the information of each of the models being tested, with each row being a model.

One Layer Models						
Name	BasicNN 1	SumLastNN 1	WNN 1	Vis 1	BasicNN 3	WNN 3
Layer 1	BasicNN	SumLastNN	WNN	Vis	BasicNN	WNN
Input	1	1	1	1	1	1
Output	1	1	1	1	1	1
Activation function	sigmoid	sigmoid	sigmoid	N/A	linear	linear
Training rate	1	1	1	0.001	1	1

Two Layer Models

Name	BasicNN 2	SumLastNN 2	SumLastNN 3	WNN 2
Layer 1	BasicNN	SumLastNN	SumLastNN	WNN
Input 1	1	1	1	1
Output 1	4	4	4	4
Activation function 1	linear	linear	linear	linear
Training rate 1	1	.1	.1	1
Layer 2	BasicNN	SumLastNN	BasicNN	WNN
Input 2	4	4	4	4
Output 2	1	1	1	1
Activation function 2	sigmoid	sigmoid	sigmoid	sigmoid
Training rate 2	1	.1	.1	1

Vis 2 The Vis 2 model has the equivalence of 4 neurons worth of computation in this setup, matching the two-layer structures.	
Layer 1	Vis
Input 1	4
Output 1	1
Activation function 1	N/A
Training Rate 1	0.001

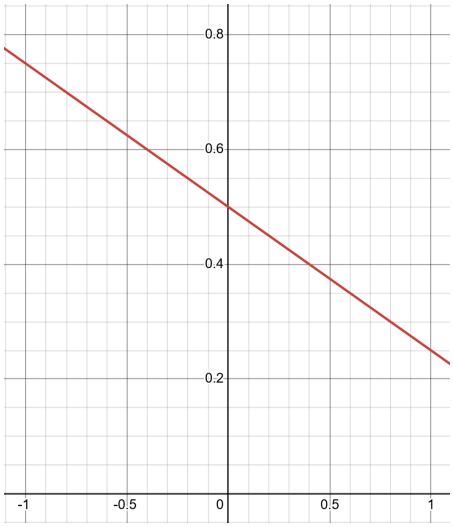
6.2 Data Generation:

For this regression test, data was generated for training and testing to ensure clear patterns without random error caused by real-world data. There are two sets of data for each

model, the training and testing set, with the training set dependent on the equation, and the test set having 1000 values on the domain of the equation. The first three equations are linear with the second three being nonlinear. Each equation has been chosen with the purpose of showing how different models perform.

6.2.1 Equation 1:

$f(x) = \frac{-1}{4}x + .5$



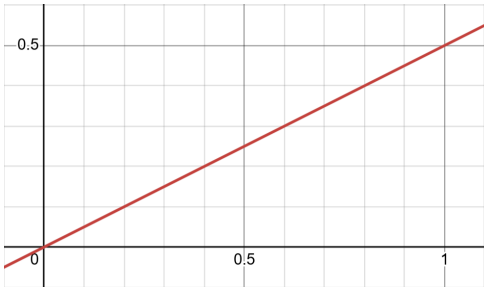
Domain:	Range:	Training Points:
$-1 \leq x \leq 1$	$\frac{1}{4} \leq f(x) \leq \frac{3}{4}$	20

This data's domain and range are set to allow every model being tested to be able to reach all the outputs easily. This function is linear, simple, and should be easy to fit so long as mathematically all the functions can actually reach the outputs. Each of the different models operates with different ranges, but all of them can handle outputs in the range of this function. This will help to test the models on even footing, which will show their raw training speed.

The training data has 20 points, which is extremely small for most AI models. This smaller data set will test the model's ability to generalize to data even with only a few points. Finding good data in a large enough quantity can be a challenge, even with the widespread spread of technology. This test will help to show what models can perform well even with only a few inputs for linear functions. This is not a perfect test as linear functions give some advantages to the models with linear activation functions, but a similar test is given later to test this with nonlinearity.

6.2.2 Equation 2:

$f(x) = \frac{1}{2}x$



Domain:	Range:	Training Points:
$0 \leq x \leq 1$	$0 \leq f(x) \leq \frac{1}{2}$	100

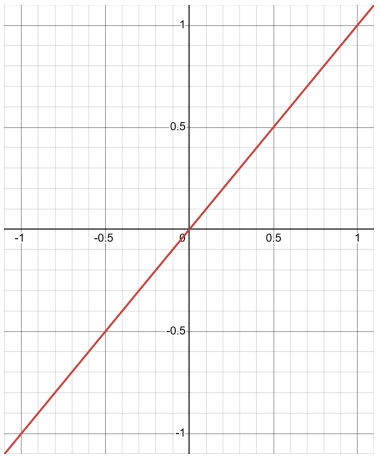
This function with the set domain will still only have positive values in a small range. The range is included in all the activation functions used, but for the sigmoid activation function, where it asymptotes at 0, it could struggle with inputs near 0.

The training data has 100 points, which should give it sufficient data to understand the linear shape of the function without being excessive. This is an attempt to show how models will

perform with an amount of data that would be expected given the pattern complexity and the size of the domain.

6.2.3 Equation 3:

$f(x) = x$



Domain:	Range:	Training Points:
$-1 \leq x \leq 1$	$-1 \leq f(x) \leq 1$	500

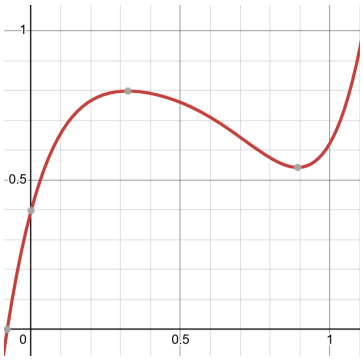
Where the domain stretches from -1 to 1, with the training and testing data being evenly distributed on this domain. This training set will have 500 data points, which is the largest linear training set. This is still a small amount of data compared to what most models would use for real-world prediction, but with such a simple equation, it should be in excess of what is needed to identify a linear pattern. This will test the model's abilities to train with larger amounts of data, which is more common than the small datasets, especially as more and more data is being collected by our technology.

This function also has negative outputs, which makes activation functions that are limited to positive outputs less likely to work. While data is easy to scale to allow any activation

function to operate, sometimes this is forgotten and can cause issues. Some models being tested can handle a very large range of output and while this is not really a benefit in most cases, it can help avoid the scaling step, which only works on bounded outputs.

6.2.4 Equation 4

$$f(x) = 10(x - .5)^5 - (x - .3)^2 + .8$$



Domain:	Range:	Training Points:
$0 \leq x \leq 1$	$\sim 0.4 \leq f(x) \leq \sim 0.8$	50

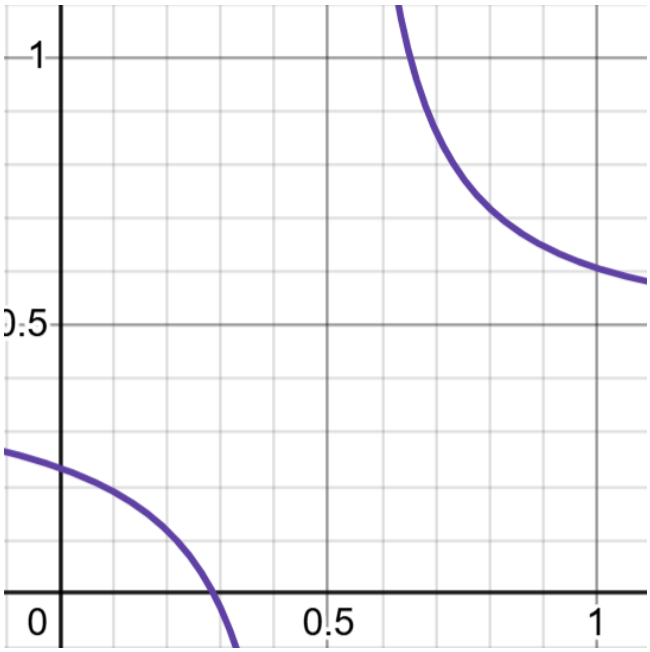
This relatively simple polynomial represents a moderately complex curve. This function is one step above that of a linear function. That said, it can be much harder to model with its varying slopes, concavities, and other properties. These additional properties will show how different activation functions handle nonlinearity. This is also much closer to a real-world data input having multiple curves and changes.

This data has very few data points in the data being used to train the model, just like in equation 1 in section 6.3.1. This small set of points is for the same reason as in equation 1,

testing the model's ability to learn with limited data. Unlike in equation one, this is much closer to a real world problem, thus it was given more data to allow the models to understand the curves.

6.2.5 Equation 5

$$f(x) = \frac{e^{sin(x-4)}}{16(x-.5)} + .5$$



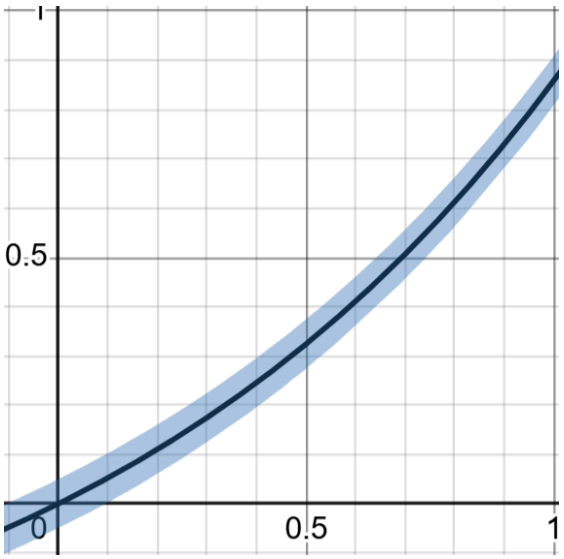
Domain:	Range:	Training Points
$0 \leq x \leq 1$	$-\infty \leq f(x) \leq \infty$	100

This equation is a more complex function with a discontinuity at 0.5. This makes it represent non-continuous functions as well as functions that have steep slopes or values going to the infinities. It is expected that most models will struggle with this type of problem as they are based on continuous mathematics that are generally bounded. In addition, the extremely large

and small numbers approaching 0.5 could cause more significant error in those regions, making the model fit just those areas and ignore the curves that are visible in the graph. This would result in not really fitting the function and just the function attempting to approach -infinity and infinity.

6.2.6 Equation 6

$f(x) = \frac{e^x}{2} - .5 \pm .05$



Domain:	Range:	Training Points:
$0 \leq x \leq 1$	$0 \leq f(x) \leq \sim 0.9$	100

This function implements a simple exponential function that has been scaled in the domain of 0-1 to be positive and less than 1 for models with the sigmoid activation function. This function will also implement random noise of a maximum magnitude of 0.05. This combination will allow for a wide range of activation functions while being more realistic with data collection, resulting in error.

Most data in the real world have some variance, either caused by instrument error during measuring data or the world just not being constant. Many factors cause data errors, making models that can not handle these variations useless in most cases. Thus, testing how models handle variance is incredibly important, and what this function is designed to test.

6.3 Methods:

There are three attributes being tested to prove that there is no one best model for every situation. The first just tests how long it takes to train using one epoch of data. The second test is the rate at which models learn per epoch, which can help to determine how well different models perform given different amounts of training time. The third is how well each model can learn the patterns of the data.

The first test uses nanoseconds to measure the time it takes for one training epoch. A training epoch is the amount of time to get through all the data in the dataset. In this case, we are using gradient descent, not sparse gradient descent, and thus use all the data before changing the weights and biases. Each model is run 100 times with 100 epochs each, with each epoch's training time being measured. The average of all the epochs across all the runs is given in a bar graph. The graph is shown for the first equation, with the rest in the appendix, as they only change in scale. This data will help to show what models are best in terms of speed, compared to other graphs, which give how good the model is at predicting the pattern.

The second test looks at how training progresses through the epochs. This is shown as a line graph for each equation with mean squared error (MSE) as the y-axis, and the x-axis being the epoch. Each model is run 100 times, and these are averaged, which helps avoid the random nature of weights and biases from affecting the results. The focus of this graph is to show the

training rate and how models do at different training stages. It also helps to see what models are doing the best in terms of MSE.

The third test is the most crude and least accurate and is simply a two-bar plot showing the model Error. One showing the final epoch of the training averaged over 100 runs of each model. The second is much less accurate but still useful, which is the error of the last trained model on the test set. The second graph shows how well each model generalizes to the functions by running the test set, which has inputs the model has never seen. The issue with this graph is that the data is only collected on one run of each of the models, with no averaging, meaning that the weights and biases randomness is not accounted for. The first graph is less useful as it only shows how the model did on data it has already seen, but is done with averages. The combination of these graphs can give a good idea of how different situations affect models differently during training, as well as the performance of these models. While these graphs are not the best, they are the easiest to understand and show that the meat model is situational based on the data given.

6.4 Results:

6.4.1 Equation 1:

The Figure 6.4.1.1 shows how each model has different calculation speeds during training. The speed of training one epoch (set of data) in a standard method for understanding the amount of energy and resources is needed to train a specific model. In this case we can see the WNN 3 (in blue) takes the least amount of computation to train, making it desirable for computers with low computational power. The models with one in their name are also doing well. This is likely because they only have a single layer, like WNN 3 and the BasicNN 3 model to its left. The reason the WNN 3 performed better is likely due to its linear activation function in

combination with its structure. The WNN 3 and WNN 1 performed better than the BasicNN of the same size and same activation function. While the WNN performed well in the speed department, it did not do as well at training.

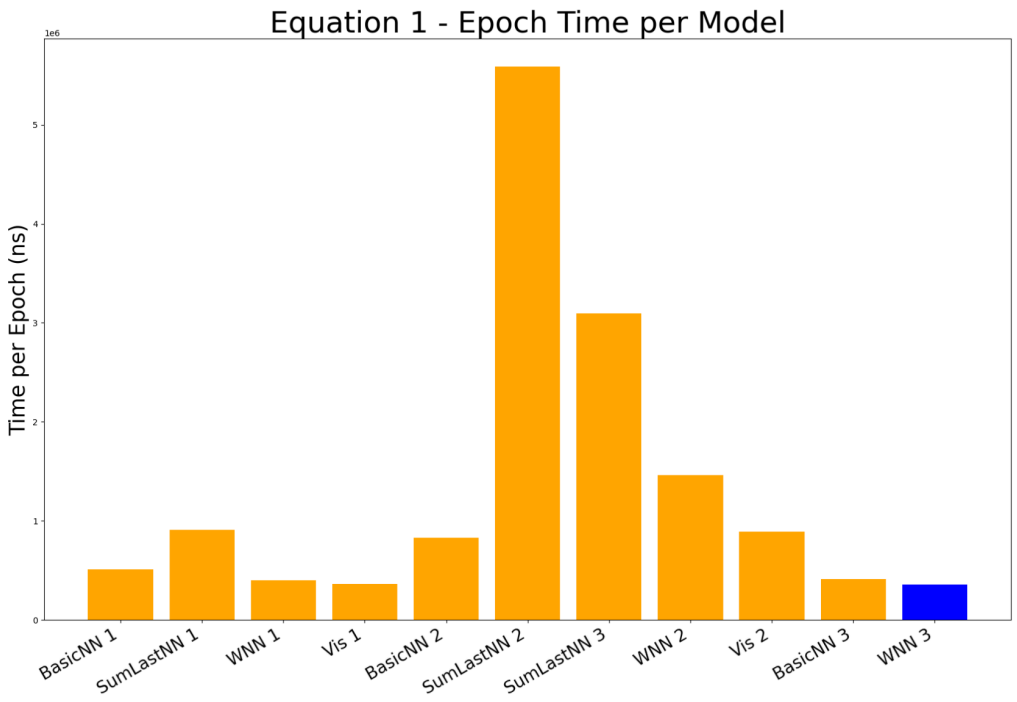


Figure 6.4.1.1 Shows which models performed the fastest computations per epoch (in blue)

While how a model can run training calculations is important, it does not matter much if it does not learn anything. The next figure, Figure 6.4.3.2, shows the progression of Error during training and the WNN 3 did not perform well. This figure is useful to identify training rates, oscillating or stagnate training, and how models perform given different amounts of time. The WNN 3 represented by the dotted cyan line improved slightly at the start but leveled out, training stagnating, resulting in many models performing better. The three leading models kept training the entire time and are the BasicNN1, BasicNN2, and the SumLastNN 3 models. These models

did well through most of the training, staying close to each other throughout the training, with the BasicNN 2 taking the lead but also apparently starting to stall at the end of training.

This type of graph can also be used to predict both what models currently are performing the best as well as how time might affect this performance. In most cases, models begin to stall and training becomes more difficult as time progresses. This is caused by the model being as close as it can be to the pattern, but this can happen at very different times, as can be seen when comparing WNN 3, which stalled early, and SumLastNN, which has not stalled in this sort of training period. In this situation, if these models were given more training epochs, the BasicNN 2 might be surpassed by the BasicNN 1 and the sumLastNN 3 models. While this does not change what model performed the best, if we were to pick a model to spend more time on, those two models that are still performing well might be worth more.

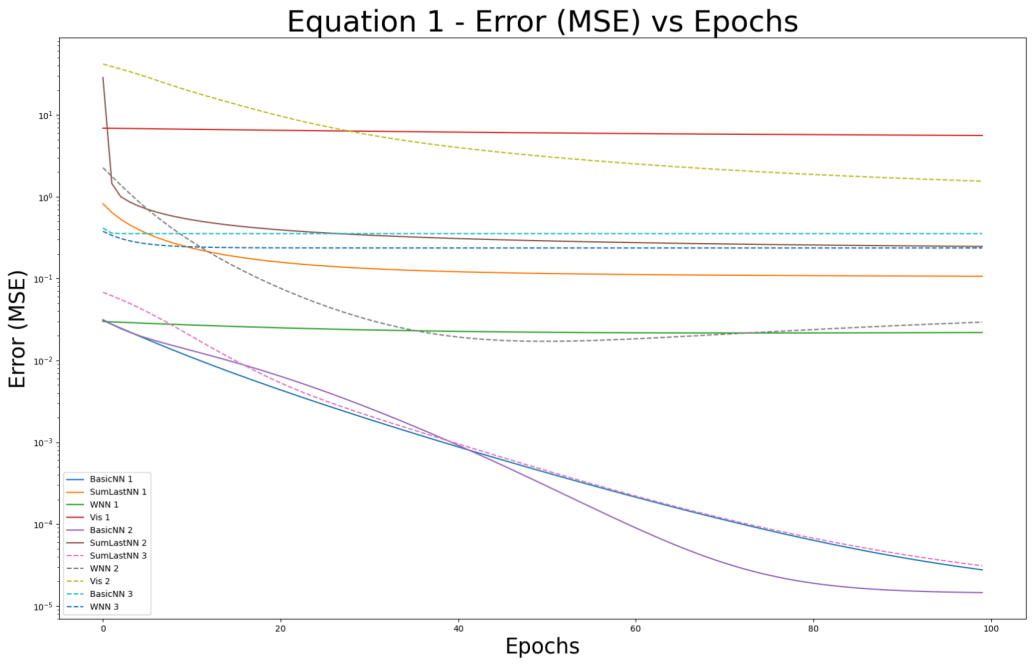


Figure 6.4.1.2 Shows the MSE per epoch during training

Figure 6.4.3.3 represents how well each model learned the training data. This graph, in connection with graph 6.4.1.4, which shows how well each model learned and identifies models that are better at generalizing curves versus memorizing the training data (overfitting). Both figures show that the BasicNN2 model performed the best, with the two runners up being the SumLastNN 3 and BasicNN 1 models, which is what was also shown in the last figure, but these are not necessarily the best models for this situation.

The SumLastNN 3 model was comparable in error to the leading model and had a steeper slope, but it is likely not worth training. While the rate at which models train and the error is important, it took much longer to train, which can be seen in Figure 6.4.1.1. Each of these graphs gives a different aspect and can not be forgotten while sometimes accuracy is the most important aspect, the ability to keep training for a long time or the speed at which a model could learn might be more important for a given situation. A language model might train for months, but a model that is learning how to walk might need to do so in only a few steps to prevent falling over. Each situation needs different attributes and this must be kept in mind while choosing the model that will be used for a situation.

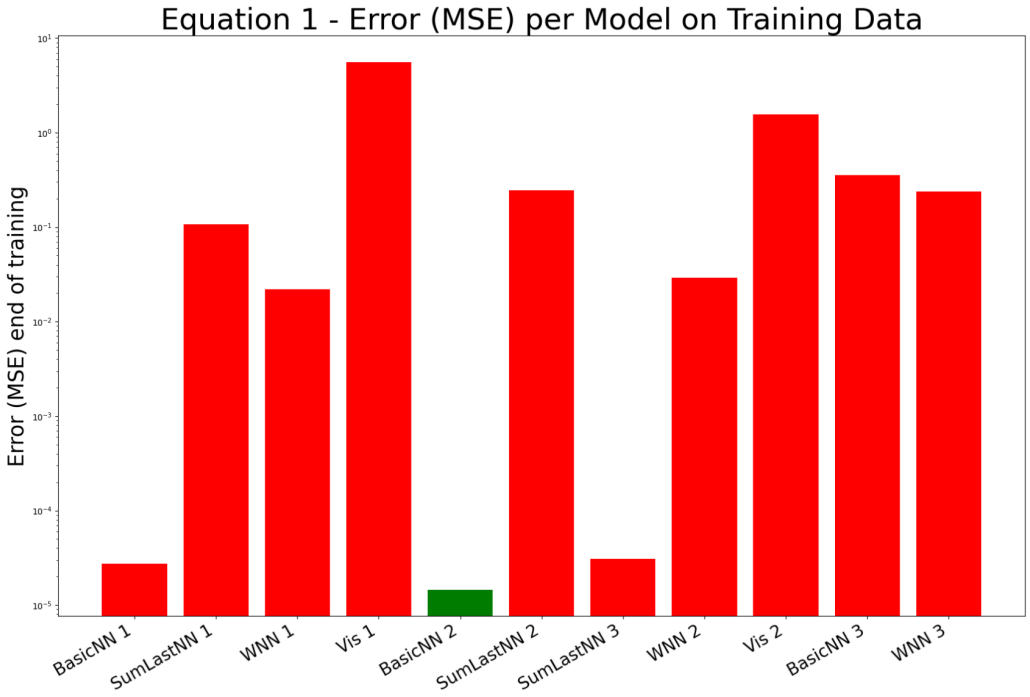


Figure 6.4.1.3 Shows the ending MSE with the training data showing what model did best fitting just the data points (in green)

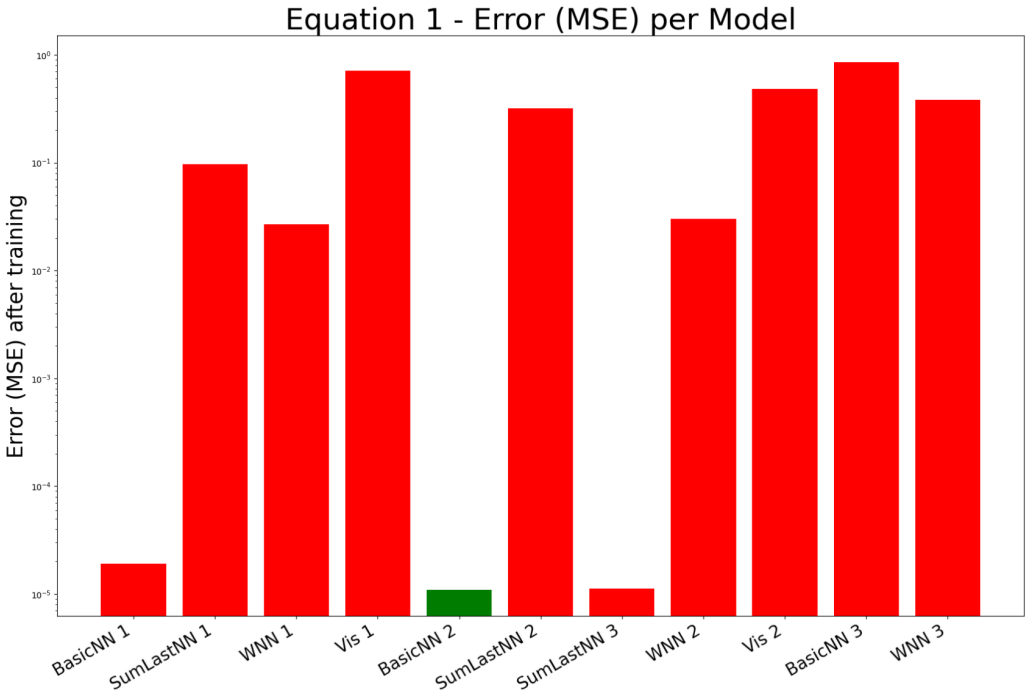


Figure 6.4.1.4 Shows the ending MSE with the testing data showing what model best fit the curve(in green)

The first test gives a chance for models with any output range restriction caused by activation function to perform well as the data was scaled to fit the models. As such, it is not surprising that the simple models with sigmoid activation functions and some of the multi-layer models are performing well. The models with more layers, like BasicNN 2 and SumLast 3, which performed the best, is likely because they have enough parameters to adjust that they could reach a linear function with ease, but in early training the BasicNN 1 did the best which can be seen if Figure 6.4.1.2 with the dark blue line. While this model did not do the best in the end, it was clearly still improving and the BasicNN 2 model had started slowing down in terms of improvement. The BasicNN 1 model was simple and probably quickly got its few parameters to match the function with some accuracy, getting passed over due to it taking longer to mitigate

the effect of the nonlinear component, due to not having as many adjustments. The multilayer models could more easily get their non-linear nature to act linearly, likely letting them take the lead over the BasicNN 1 model soon after training started.

In this situation, the BasicNN style model performed the best with low error according to the models. While the BasicNN 2 performed better in terms of error, the BasicNN 1 model takes less time to train per epoch and is likely to be able to keep training. This makes the choice between these models situational, with the BasicNN 1 better for situations where speed of training matters, and the BasicNN 2 for situations where slightly slower training speed is acceptable and low error is crucial. This backs the claim that model selection is situational, but only slightly, as they are very comparable models in most aspects

6.4.2 Equation 2:

Equation 2 tested a more significant number of inputs, and this has affected how fast models could calculate backpropagation. This can be seen in Figure 6.4.2.1, which can be found in the appendix, which shows that the increase in training data size increased the time per epoch compared to 6.4.1.1. The modes' speeds relative to one another are about the same as in Figure 6.4.1.1, with only the scale on the side of the graph showing any differences. This means that the most efficient models are still efficient given more data. This increase in cost can not be easily mitigated with the model choice, but with larger sets of data, considerations like picking simpler models might be warranted.

The figure showing how well each model is training over time, Figure 6.4.2.2, shows us that BasicNN 2 is doing very well, just as with Equation 1. Compared to how the models did with equation 1, equation 2 has much more competition between models, with the gap between the best-performing models and the rest much smaller. The three leading models for equation 1

started to level out much faster with equation 2 keeping all the models closer. This is likely due to the sigmoid activation function on the three leading models struggling with equation 2's y intercept at 0. While the same models performed the best, the competition was much closer and the SumlastNN 3 did better than the BasicNN 1 model on this equation and appears to be close to overtaking the BasicNN 2 model. But there are some interesting developments when looking at how the error changes when looking at the test dataset.

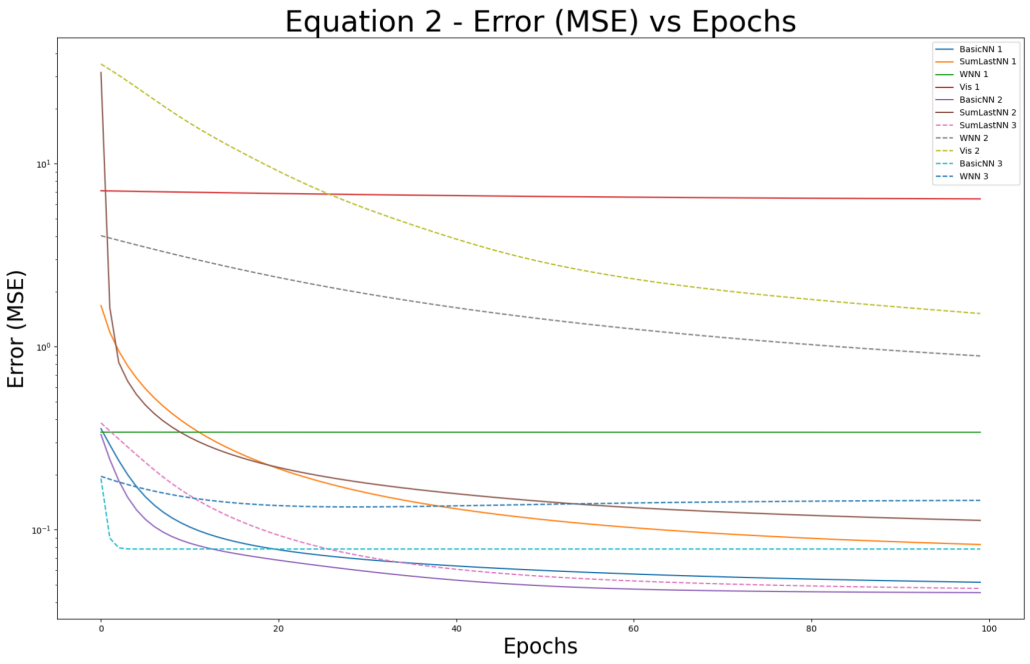


Figure 6.4.2.2 Shows the MSE per epoch during training

Figure 6.4.2.4 shows some interesting changes compared to the results with the training data in Figure 6.4.2.3, with most models changing their error. When running the test set on the last trained model, the BasicNN 3 model, in 4th place in terms of error during training, is now the best performing. This equation training caused some major issues with overfitting, with the top three models struggling with generalization, having memorized the data. The second place model on the test, the WNN 3, is also the fastest at back propagation, and is now worth more

consideration as a model, while not apparently training well, it ended up with very low errors with the test set.

The test set revealed that a linear activation function, while not appearing well during training, has benefits. The models that overfit the training data used the sigmoid activation function while the models that performed well at generalizing the curve used a linear activation function. It is surprising that these linear activation functions did not perform well during training as this is what they are designed for, but apparently they still held up with the test set.

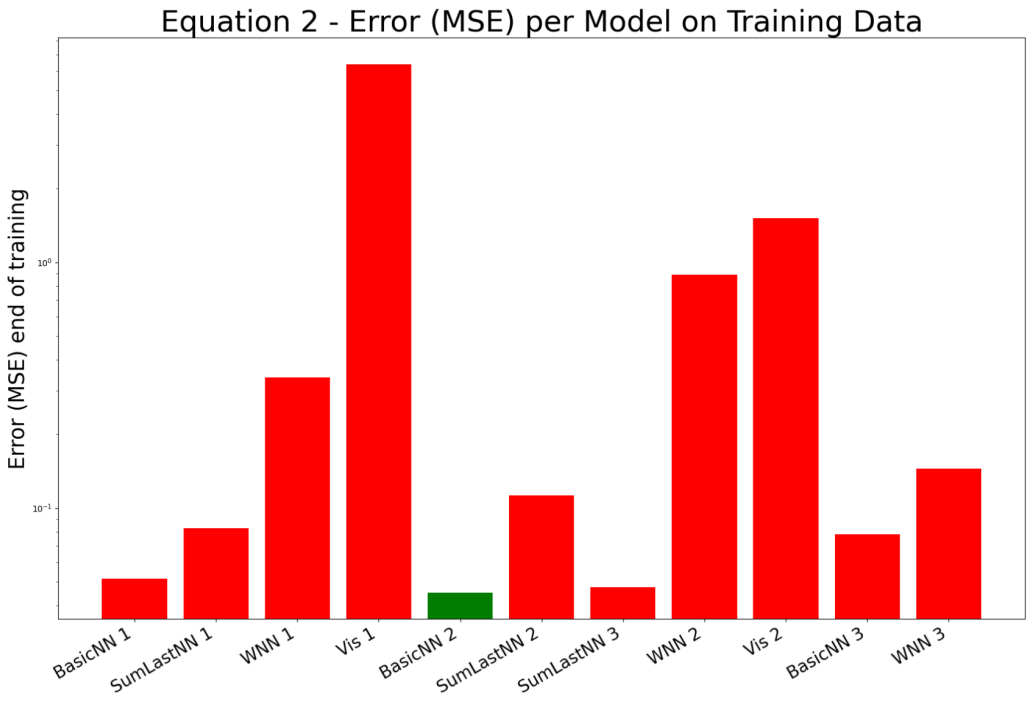


Figure 6.4.2.3 Shows the ending MSE with the training data showing what model did best fitting just the data points (in green)

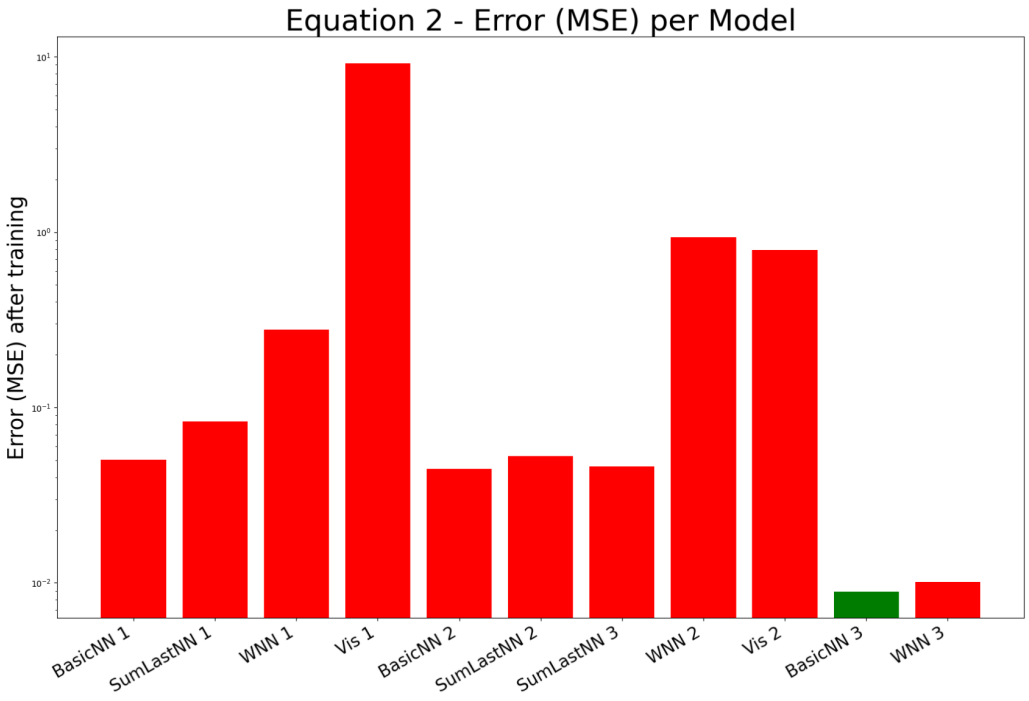


Figure 6.4.2.4 shows the ending MSE with the testing data showing which model best fit the curve(in green)

Equation 2 showed that the activation function is extremely important when selecting a starting model. The starting model has many properties that can be adjusted, but the activation function is one of the most commonly adjusted, and for good reason. While the focus of this paper has been on the neuron equation, the activation function allows nonlinearity in neural networks, and the properties of the function shape the outputs and patterns the models can fit. In this case, the linear activation function allowed models to avoid overfitting the data, resulting in a model that was much closer to the actual function. While during training it appeared not to have the best error, the linear nature of the function kept it close to the actual equation attempting

to be fit. And while this is impressive, the activation function is hard to practically adjust in most scenarios, while in this case, we know the pattern; in most situations, it is impossible to visualize with thousands of dimensions, making the activation function choice much harder.

While extensive activation function testing on large problems might take too long, checking what models operate the fastest does not. The second-best performing model, the WNN 3, did extremely well with the test set and takes the least computation to calculate. This is the first time this model has performed well so far, but it is efficient, and this efficiency is much easier to find than selecting the perfect activation function. While efficiency of operation does not matter if it does not learn, this model can learn, and with its efficiency, might be the best model in this situation. This further shows that there is no one best model, as this model did not perform well with the first equation.

6.4.3 Equation 3:

Equation 3 again increases the number of data points given during the training, with similar increases in the time for each Epoch per model. This can be found in Figure 6.4.3.1 in the appendix. The change was virtually the same as the difference between equations 1 and 2. Equation 3 also has some shared results in terms of model performance.

BasicNN 3, one of the best models during the testing of equation 2, performed extremely well on this test. BasicNN 3 with a linear activation function and simple neuron equation allowed it to model equation 3 extremely well. The other models that performed well during the testing of equation 1 and training of equation 2, like the BasicNN 2, have a sigmoid activation function,

which could not handle the negative outputs of equation 3. The BasicNN 3 model does not have these limitations, allowing it to perform extremely well

The BasicNN 3 model trained extremely fast and had very little competition, as can be seen in Figure 6.4.3.2. The BasicNN 3 model learned extremely fast at the start and leveled out quickly. While it did stop learning very fast, only the Vis 2 model appears to still be learning after the halfway mark. Equation 3 and Figure 6.4.3.2 show how the data given can affect how models perform and how some models just outright fail. The linear activation function is not commonly used, with the ReLU activation function being more common, as it actually has a chance of changing data, but it also does not have negative outputs. Things like range and domain must be kept in mind when picking starting models, or your error will be very high.

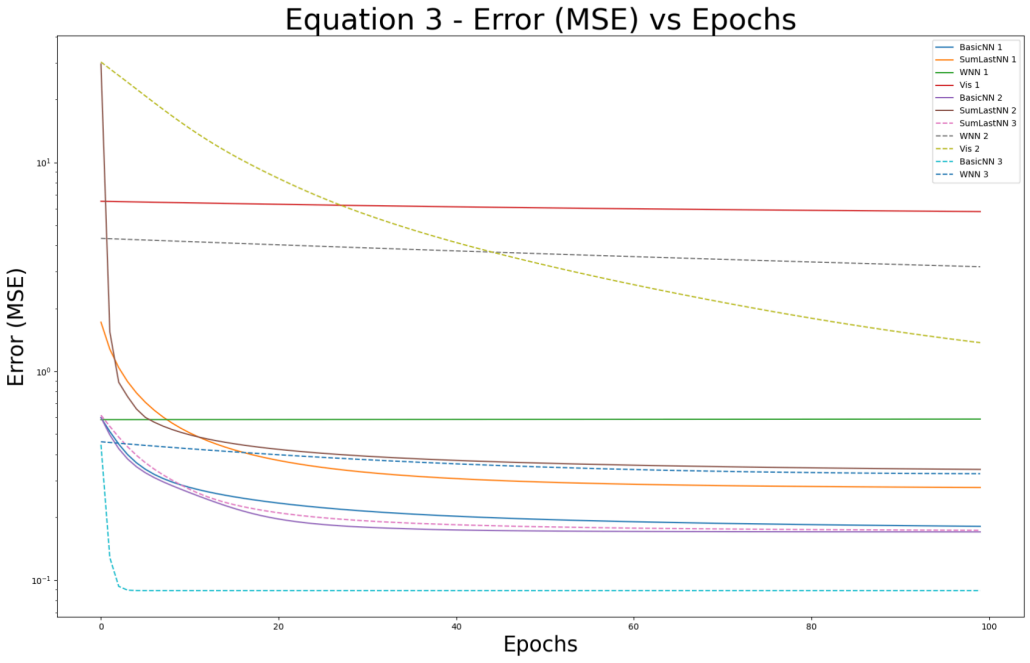


Figure 6.4.3.2 Shows the MSE per epoch during training

As shown in the previous figure, BasicNN 3 performed extremely well. Figure 6.4.3.3 shows that while it is clearly the best model, there are three models that, while worse, are comparable to one another despite different mathematical models. BasicNN 1, BasicNN 2, and SumLastNN 3, while not optimal, do show that some models perform similarly despite large differences in the neuron equation and structure. This paper has been working on showing that the math that builds models can help with starting model situations, as no model works for every scenario, and while this is true, the math being used and described works with so many dimensions that it can be hard to identify. While the BasicNN 3 model's performance makes sense due to its activation function, the similarities between other models are harder to determine. This makes model selection even harder because even if the math looks different, they could perform the same in terms of error.

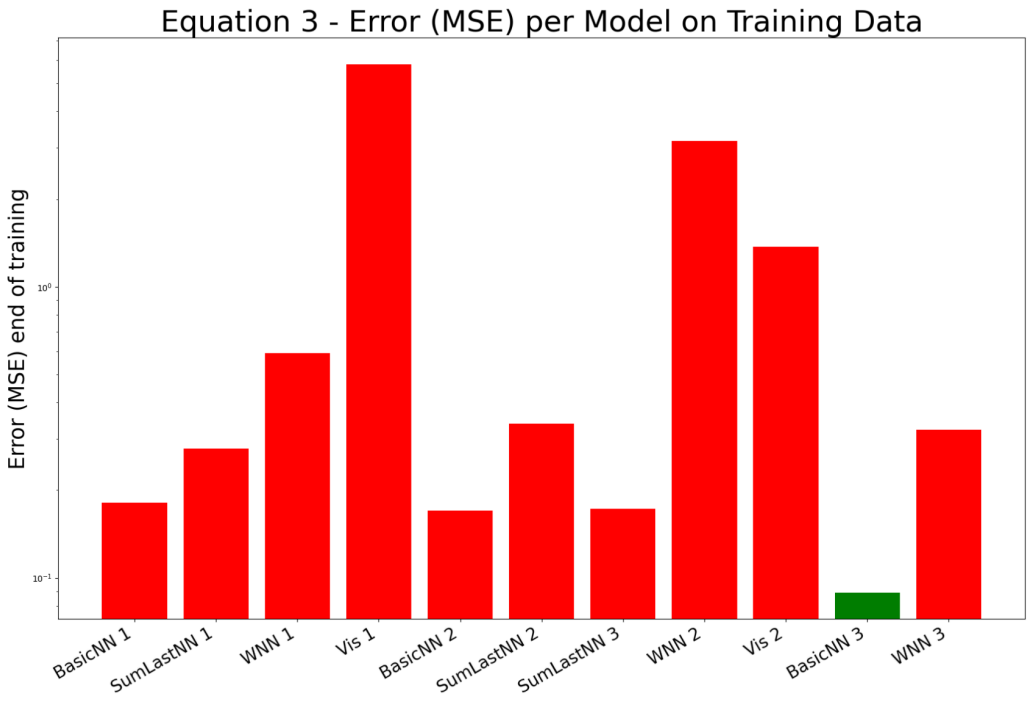


Figure 6.4.3.3 Shows the ending MSE with the training data showing what model did best fitting just the data points (in green)

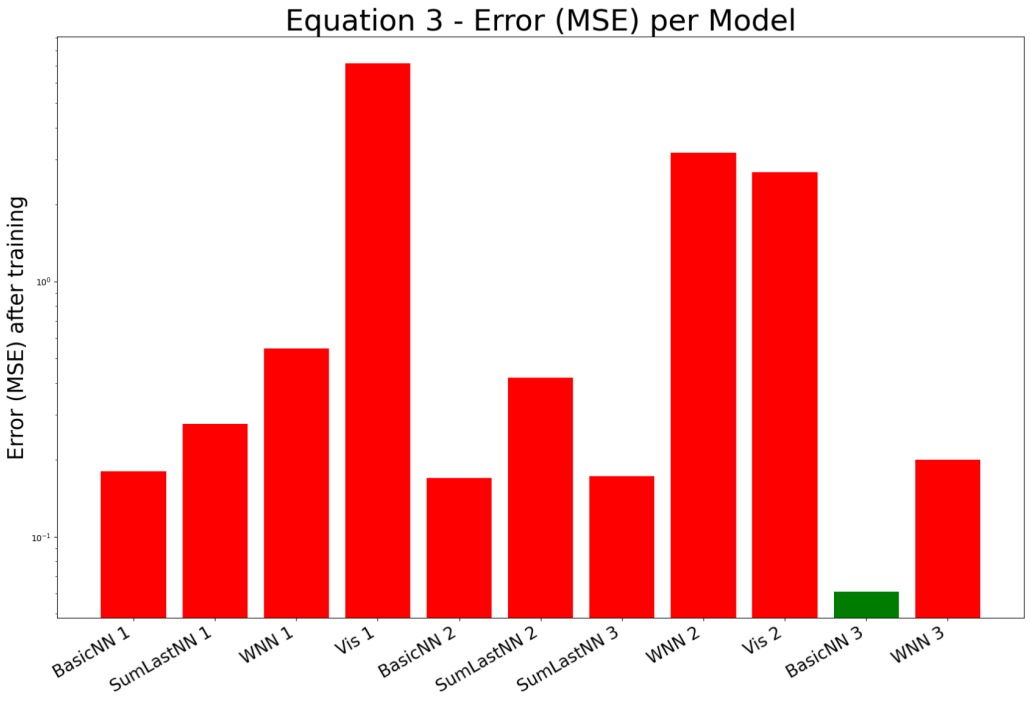


Figure 6.4.3.4 shows the ending MSE with the testing data showing which model best fits the curve(in green)

Equation 3 was meant to show that the range of the function being modeled can affect what models can get good results. This can be seen in Figure 6.4.3.4, which shows that BasicNN 3 performed the best with a linear activation function. A linear activation gives the model the ability to output any value, unlike the sigmoid function, which can only output positive values. Examining the BasicNN 2 and BasicNN 1 model in Figure 6.4.3.3, it can be seen that they did not perform as well with their sigmoid activation functions; in fact, their error was nearly the same, meaning they both had trained the best they could to the function given their activation function.

The WNN 3 model also operates with a linear activation function but did not perform well. This shows that it is not just about the activation function and that the BasicNN with its simple neuron equation has benefits. It also shows that the activation function, while important in the starting model, is not the only factor when creating a good model. This is important as while the past results have shown that starting model matters, especially with the activation function, this shows that other factors matter as well.

6.4.4 Equation 4:

The training of equation 4 is very interesting, as about halfway through the training the model that is performing the best switches. The BasicNN 3 model performs extremely well in the beginning and then levels out, which has been a common occurrence with this model. But the Vis 2 model catches up as it does not level off as fast, resulting in it overtaking the other model just past the halfway mark. This makes this particular test extremely interesting as depending on the amount of time for training, different models perform the best.

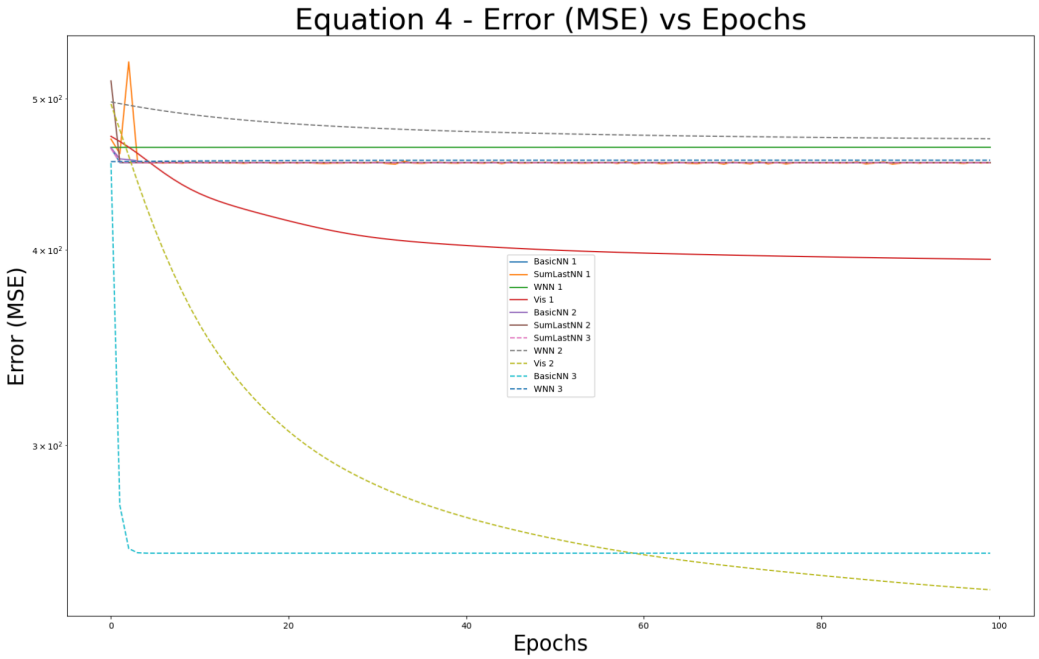


Figure 6.4.4.2 Shows the MSE per epoch during training

This equation also is the first time the Vis 2 model has performed well. Being nonlinear, this makes some sense as the Vis 2 model is heavily based on the sine function. Figure 6.4.4.3 and Figure 6.4.4.4 show that it performed well in both the training and testing, and Figure 6.4.4.2 shows a steep slope near the end of the training, meaning that the vis model is likely to continue to improve and even further outperform the other models.

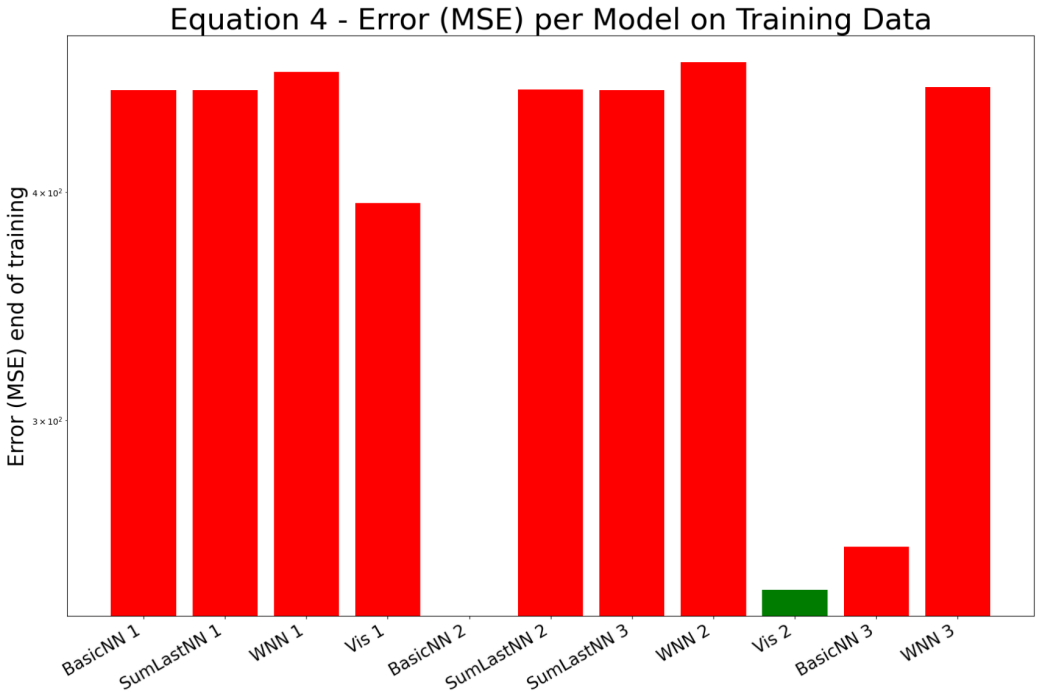


Figure 6.4.4.3 Shows the ending MSE with the training data showing what model did best fitting just the datapoints (in green), Note BasicNN 2 failed during training and has no bar

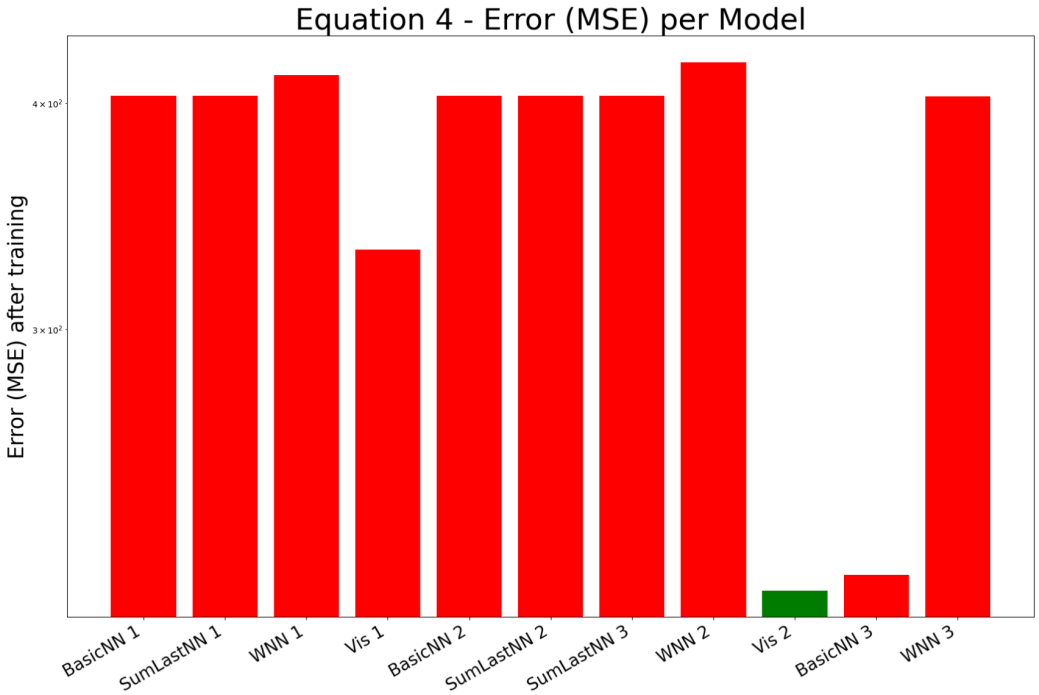


Figure 6.4.4.4 Shows the ending MSE with the testing data showing which model best fit the curve(in green)

If we take a look at the function generating the data, these results make some sense in regards to the Vis 2 model's performance. The function, while not related to the sine function, does have characteristics that resemble a sine or cosine function. The alternating concavity makes the sine base Vis 2 model perform understandably well. The Vis 1 model is also based on the sine function, but there is not much that can be done with a single sine function as the entirety of its network. The second best performing model is the BasicNN 3 with a fully linear activation function. It might be surprising that it did well in this composition, but when looking at where it performed the best, it makes sense. Figure 6.4.4.2 shows that the training all happens at the beginning, meaning it just puts a line on the graph. This line is much easier to train, resulting in a fast approximation, but without any ability to curve it, stopped improving. This is unlike the Vis 2 model, which took its time, but has the ability to match the function's curves given enough time.

6.4.5 Equation 5:

All the models performed similarly on equation 5, which is likely one of the hardest equations. This equation had an asymptote reaching toward infinity and negative infinity, making it extremely hard to fit this function. The Vis 2 model is doing the worst, which is interesting as the equation generating the data has a sine function, but apparently it is so buried in the equation it did not help. Most of the models performed similarly but had very little slope, meaning they are not improving. Given enough time, the WNN 2 and Vis 2 model may reach the computing

models as they are still learning. This equation gave most models trouble but did result in most models ending with similar errors.

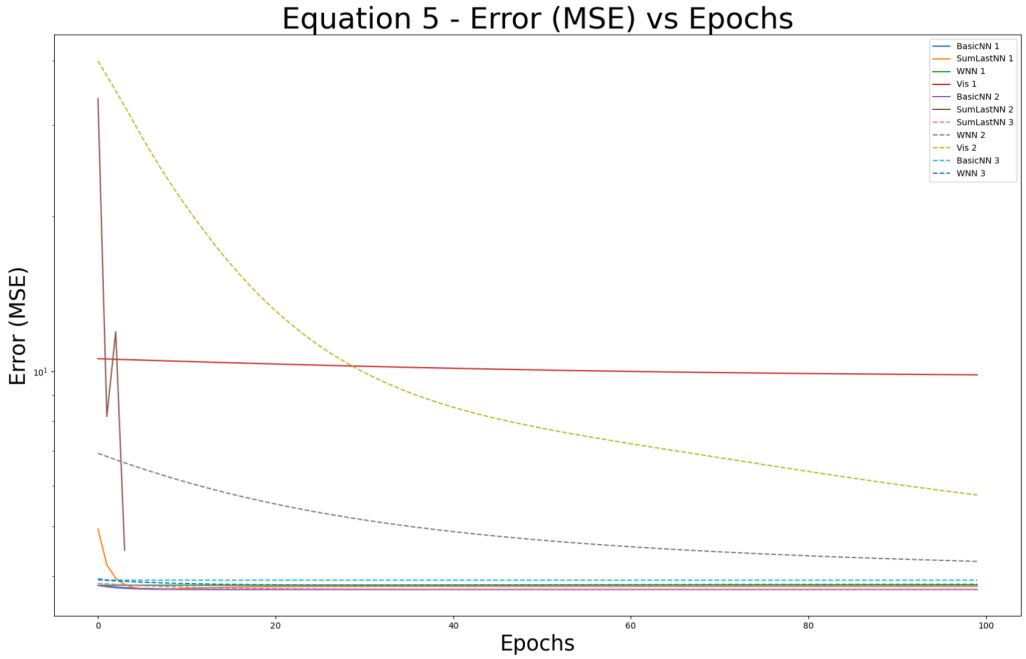


Figure 6.4.5.2 Shows the MSE per epoch during training

While none of the models did very well in terms of error, they were very close to each other. Most of the models had an error as seen in Figure 6.4.5.3, of just under 4, with the log scale being basically linear, as all the models are very similar. The three exceptions are the Vis models and SumLastNN 2, with the Vis models having high error, and the SumLastNN 2 model catching an error, resulting in data being lost. This can be seen in the above graph where the brown line cuts off. This does not mean it did not train on every iteration, as can be seen in Figure 6.4.5.4, where it performs the same as the rest of the models at least on the last run. Most of these models did not learn much, but the similarities between the models are interesting, as all the models operate with different equations.

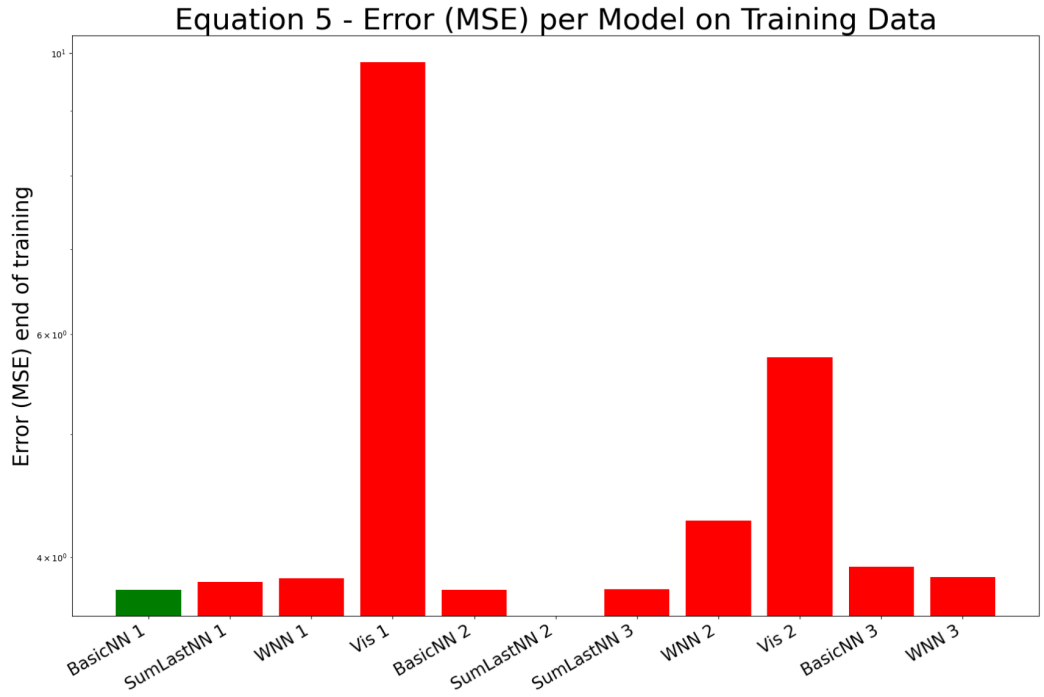


Figure 6.4.5.3 Shows the ending MSE with the training data showing what model did best fitting just the data points (in green)

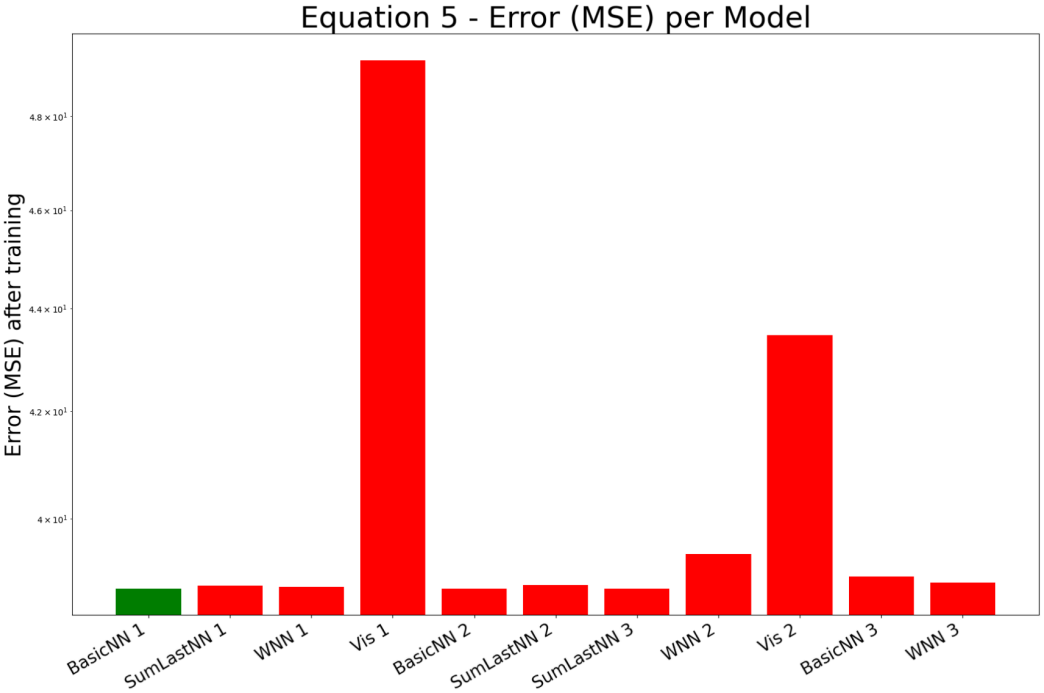


Figure 6.4.5.4 Shows the ending MSE with the testing data showing which model best fit the curve(in green)

Equation 5 has a vertical asymptote to infinity and negative infinity, which none of these models can duplicate, likely resulting in all the models struggling and ending with high errors. No models stood out, and while the BasicNN 1 model did perform the best, most other models did similarly, and thus, while this is likely because of the sigmoid activation function being able to hit some of the early curve, it is hard to say. As there is not much difference, there is no way to easily conclude why this model performed any better than the other models.

6.4.6 Equation 6:

This is the first time these models are seeing random variations, which have caused some changes to Figure 6.4.6.2. Unlike past tests, the Error vs Epochs graph is not smooth, showing that the variation in data causes fluctuations during training. This goes to show that good data is important, as when there are fluctuations, the pattern becomes harder to identify, and training is a less smooth process. But as the real world does not lead to naturally clean data, it is important to test how models handle this type of training. Interestingly, Vis 2 does not show this variance, likely because of the smaller training rate, and while it is not performing well, it is clearly still improving, while the WNN 3 model and the BasicNN 3 model are progressively getting worse. This likely means that these setups with a linear activation function struggle when the data becomes inconsistent.

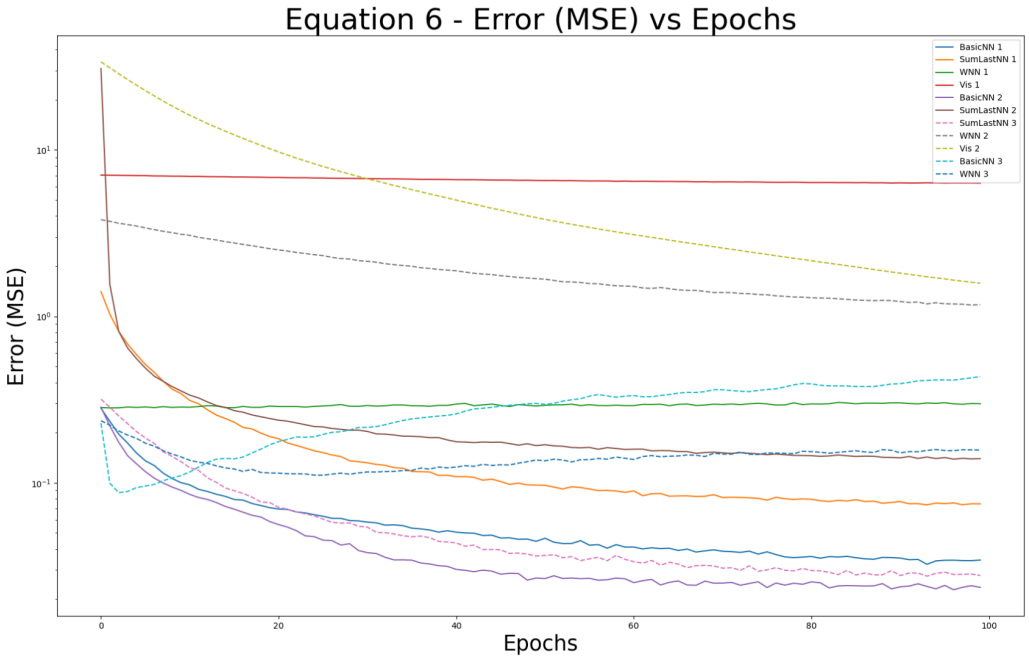


Figure 6.4.6.2 Shows the MSE per epoch during training

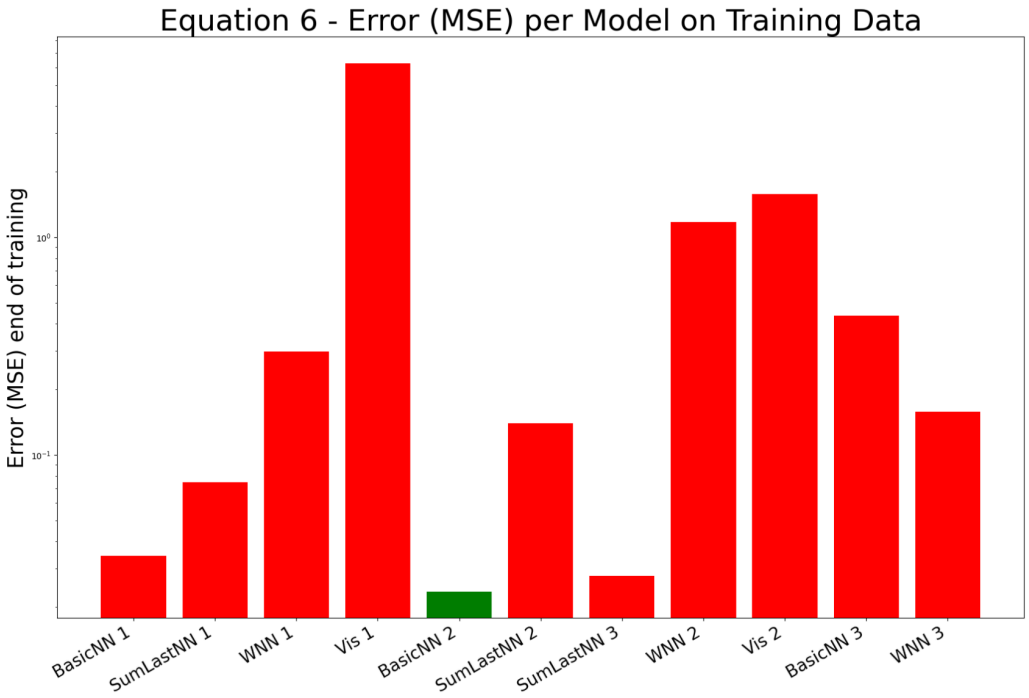


Figure 6.4.6.3 Shows the ending MSE with the training data showing what model did best fitting just the data points (in green)

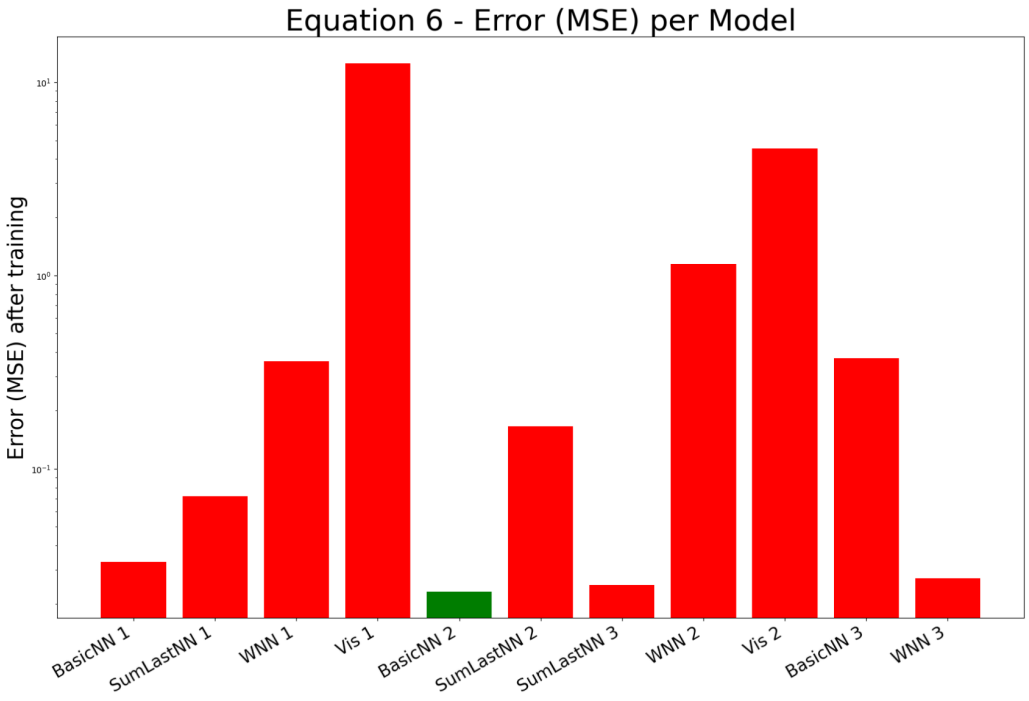


Figure 6.4.6.4 Shows the ending MSE with the testing data showing which model best fits the curve (in green)

While the linear activation function did not handle this data well, the BasicNN 2, one of the most sophisticated models with two layers, performed the best. The BasicNN 2 and the SumLastNN 3 models, which are the two best models, use two layers with a sigmoid activation function. Both these models have many parameters that can be adjusted and appear to handle randomness well. Model complexity slows down how fast a model can train, which can be seen in Figure 6.4.6.1, in the appendix, but they are also better at finding patterns due to the more options for adjustment. It should be noted that while these more complex models performed better on this model, the WNN 3 model performed well with the test data on the last trained model and is more efficient. The complexity of the two best models is likely why they

succeeded, but the cost of speed and power might make the WNN 3 a better pick for this situation.

6.4.7 Combination of results from all equations:

While there were some models that outperformed others, there is variation in what models perform the best. This proves that there is no best model for every situation and that the proper selection of starting models can improve efficiency and how well the model can train. While taking the time to properly set up the starting model to best model the resources and data of a given situation, the below table gives some ideas of what models will generally be better. If you just look at the best performing models, the Basic models are generally better when it comes to error for regression tasks. The Vis 2 model is better for functions with multiple concavities, with the WNN 3 model being good in situations where there is not much computation possible. While the starting model is situational, there are some models that generally operate better than others.

Equation	Best MSE Model for Test data	Best MSE Model for Training	Worst MSE Model for Test data	Worst MSE Model for Training	Fastest Computation per Training Epoch
1	BasicNN 2	BasicNN 2	Basic NN 3	Vis 1	WNN 3
2	BasicNN 3	BasicNN 2	Vis 1	Vis 1	WNN 3
3	BasicNN 3	Basic NN 3	Vis 1	Vis 1	WNN 3
4	Vis 2	Vis 2	WNN 2	WNN 2	WNN 3
5	BasicNN 1	BasicNN 1	Vis 1	Vis 1	WNN 3
6	BasicNN 2	BasicNN 2	Vis 1	Vis 1	WNN 3

While the above table is good for most applications, each column only looks at one figure, which is a small picture of what is truly happening. The below table shows more arguable results based on analysis of the combination of graphs. Overall, it is a general look at all the graphs and is the same as how the models performed on the test data, as most of those models were relatively sound in most categories. Best performance looks at error and how long each epoch takes to run, and is what would likely serve best if computation is a concern. Unlike just computation like the last row of the previous table, it takes into account error and each of the models was competitive in this category. The final two categories look at how training progresses over time and can be used to identify models that train well initially, which can be useful if training time is limited.

Equation:	Overall	Best for Performance	Best for Short training (epochs<10)	Best for medium training (epochs<50)
1	BasicNN 2	BasicNN 1	BasicNN 1	BasicNN 2
2	BasicNN 3	WNN 3	BasicNN 3	BasicNN 2
3	BasicNN 3	BasicNN 3	BasicNN 3	BasicNN 3
4	Vis 2	BasicNN 3	BasicNN 3	BasicNN 3
5	BasicNN 1	WNN 3	Unknown	Unknown
6	BasicNN 2	WNN 3	BasicNN 3	BasicNN 2

While this data is biased toward the BasicNN models, there is variation, with the WNN 3 model performing well. The starting models are essential with the neuron equation, benignly important, but less so than the activation function and structure, as shown by how the variation of the BasicNN model performed differently in different situations. The neuron equation appears to still be able to be improved upon at least situationally, as shown by the Vis 2 and WNN 3

models, but the focus of future research should be in methods of identifying structure and activation functions, at least for regression-style problems.

7 Part VI: Experiment 2 Methods and Results

While the past tests focused on the model with a single input and output, most situations operate in higher dimensions. The most common example and the second experiment is image classification. Image classification tests both the model's ability to classify and the ability to process multiple inputs. Several of the different models discussed thus far are designed or have flaws around the use of more inputs. More inputs normally mean more weight, biases, and overall connections, which can slow things down. The models are thus different, some being left out due to not operating properly, with the two CNN models being added as they only really operate on larger inputs.

7.1 Models:

Unfortunately, we are likely not to see results from any of these models that compare to other published results due to inefficiencies in the code and the additional features often implemented in library-based models like momentum. This said it also means that this will be a better representation of how these models work in their most basic state and the raw potential before improvement.

Two-Layer Models						
Name	BasicNN 1	SumLastNN 1	WNN 1	Vis 1	BasicCNN 1	ExponentialCNN 1
Layer 1	Flaten2D	Flaten2D	Flaten2D	Flaten2D	BasicCNN	ExponentialCNN

Input 1	(28,28)	(28,28)	(28,28)	(28,28)	(28,28)	(28,28)
Kernel 1	N/A	N/A	N/A	N/A	(28,19)	(28,19)
Output 1	784	784	784	784	(1,10)	(1,10)
Activation function 1	N/A	N/A	N/A	N/A	sigmoid	sigmoid
Training Rate 1	N/A	N/A	N/A	N/A	0.01	0.01
Layer 2	BasicNN	SumLastNN	WNN	Vis	Flaten2D	Flaten2D
Input 2	784	784	784	784	(1,10)	(1,10)
Output 2	10	10	10	10	10	10
Activation function 2	Sigmoid	Sigmoid	Sigmoid	N/A	N/A	N/A
Training Rate 2	0.01	0.0001	0.0001	0.0001	N/A	N/A

Three Layer Models			
Name	BasicNN 2	BasicCNN 2	ExponentialCNN 2
Layer 1	Flaten2D	BasicCNN	ExponentialCNN
Input 1	(28,28)	(28,28)	(28,28)
Kernal 1	N/A	(9,9)	(9,9)
Output 1	784	(20,20)	(20,20)
Activation function 1	N/A	sigmoid	sigmoid
Training Rate 1	N/A	0.01	0.01
Layer 2	BasicNN	BasicCNN	BasicCNN
Input 2	784	(20,20)	(20,20)

Output 2	400	400	400
Activation function 2	sigmoid	N/A	N/A
Training Rate 2	0.0001	N/A	N/A
Layer 3	BasicNN	BasicNN	BasicNN
Input 3	400	400	400
Output 3	10	10	10
Activation function 3	sigmoid	sigmoid	sigmoid
Training Rate 3	0.0001	0.001	0.01

7.2 Data:

Unlike in the first two tests, this model will not be trained on non-generated data through the use of an equation. Instead, we are using one of the most well-known datasets, MNIST, which gives 10,000 examples of 10 digits 0-9, examples of which can be seen in Figure 7.3.1. These are handwritten and pre-labeled, both making this a close to realistic problem while being easy to implement and test. For this test we are using an adaptation of the set by Dato-on, who converted the data into a more usable format(Dato-on).

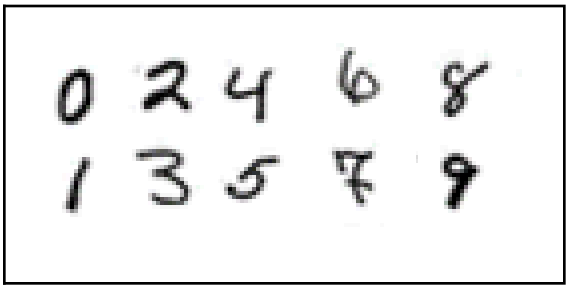


Figure 7.3.1 shows a subset of the dataset being used to train the models (Dato-on)

7.3 Methods:

This test follows almost the exact same method as in experiment one, described in section 6.4. The only difference is that instead of epochs, we are looking at batches. A batch is a subset of the data, which allows models to train faster as there is less computation before the model adjusts its weights and biases. In this case, we are using a batch size of one, which is generally a horrible choice. A batch size of one means that each batch has just one datapoint, meaning that the model is adjusting to match one input at a time, which can easily result in overfitting. Unfortunately, the time it takes to run larger batches is extreme, and with a smaller learning rate, much of the issues can be mitigated. When training models for operation, the batch size should be much larger as this helps with finding general patterns and connections in data.

The rest of the test is identical, with each model being run 100 times and running 100 batches of data. Each batch of data is randomly selected from the dataset, but each model is given the same batches of data to ensure fairness. The graphs, apart from the more generalized graph done on the test data, are averages over the 100 runs and batches, if applicable, with the test set run on the last trained model.

7.4 Results:

This higher-dimensional real-world data is sure to have very different results from the first experiment. The first of which can be seen in Figure 6.5.6.1, which shows the training speed of these models. All the other graphs of this nature looked very similar up to this point, with this

graph having to be switched to a log scale to properly compare these models. There is a general increase in computation time, which is to be expected given the increase in input size. In addition, the complexity of the models generally increased.

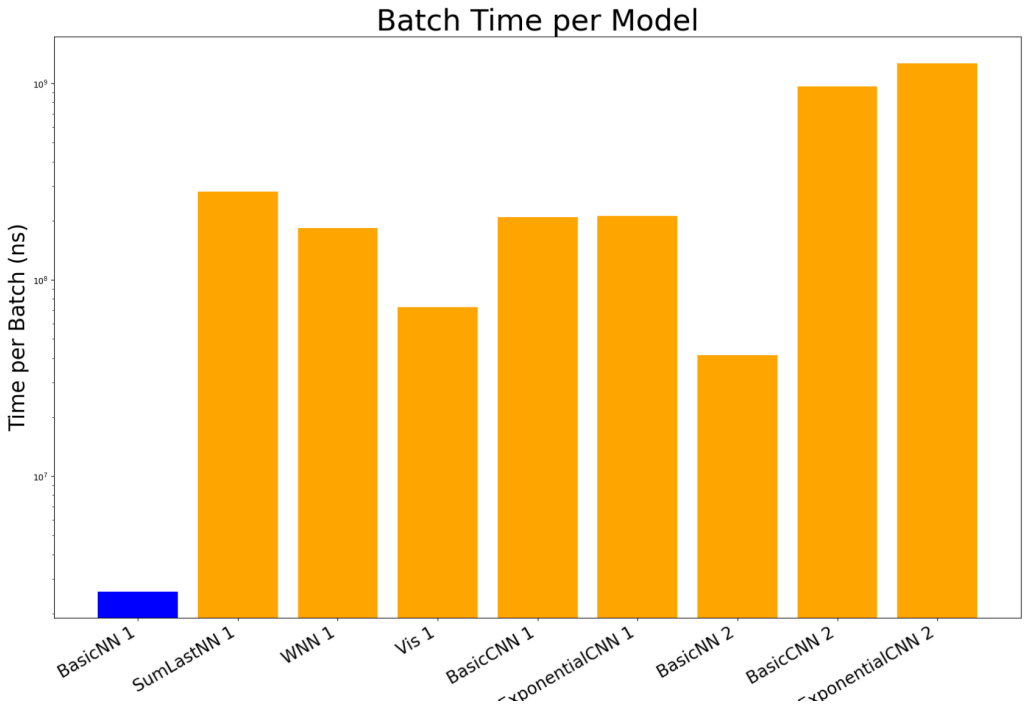


Figure 6.5.6.1 Shows which models performed the fastest computations per batch

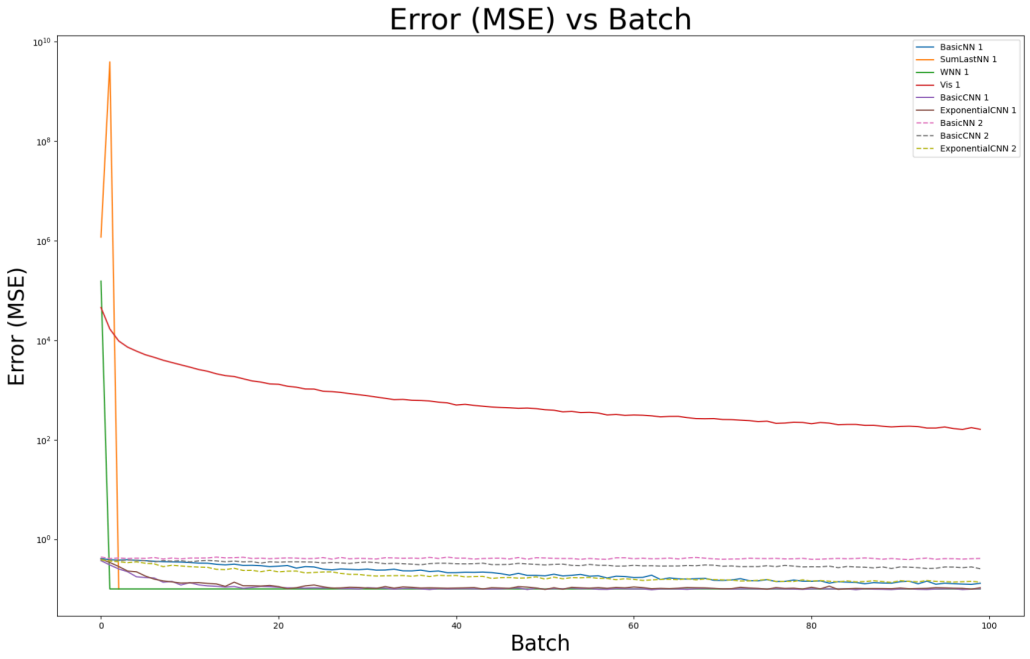


Figure 6.5.6.2 Shows the MSE per batch during training

This is the first real data training scenario for these models tested in this paper. The rest of the tests were done through generated data with very little variation or oddities. This data is true handwriting and a much better test of these models. Figure 6.5.6.3 shows a much more common training graph with much higher variation compared to most other tests. This is often fixed with slower training rates which were only partially implemented in this test. Several models are shown in this same figure that are still improving at a rapid rate and could potentially pass some of the other models given enough time.

Figure 6.5.6.3 Shows the MSE per batch during training but zoomed in to the best models

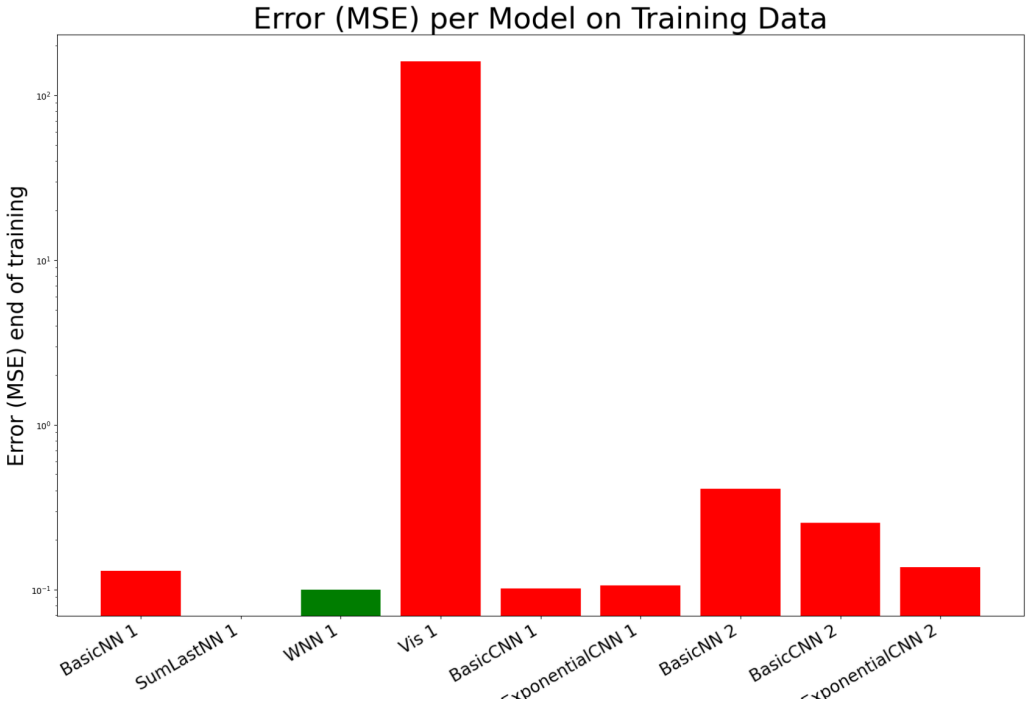


Figure 6.5.6.4 Shows the ending MSE with the training data showing what model did best fitting just the datapoints (in green), note SumLastNN 1 encountered an error during training and has no bar

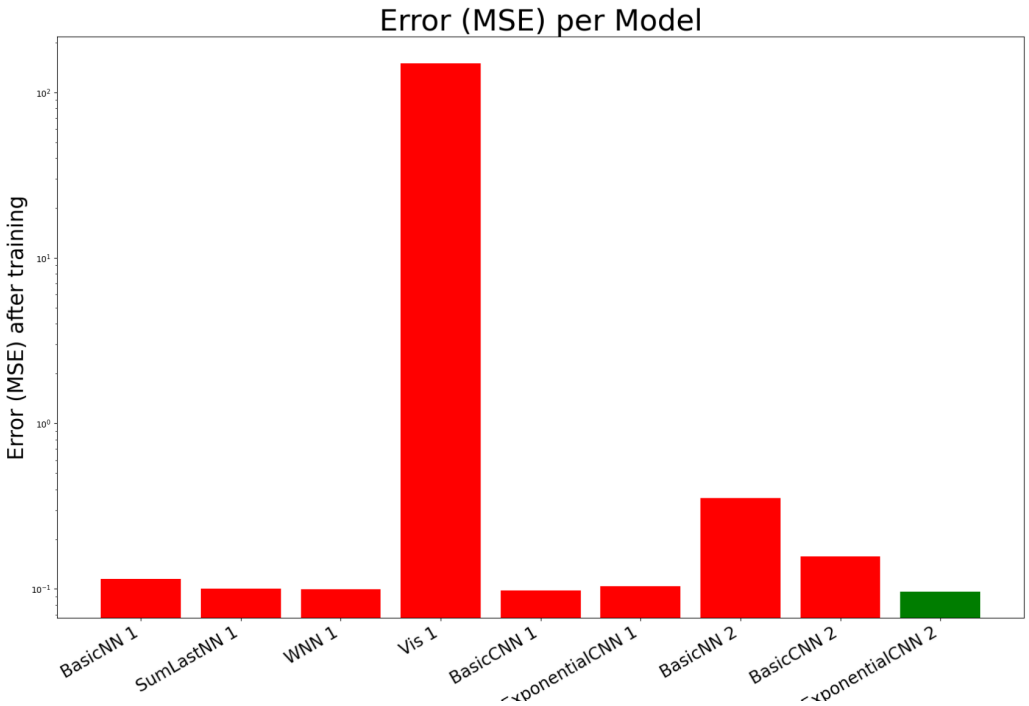


Figure 6.5.6.5 Shows the ending MSE with the testing data showing which model best fit the curve(in green)

This classification test further shows how experimental models are worth looking into even if they do not work for every situation. The exponential CNN, while not useful in regression tasks, performed extremely well during this test, as can be seen in Figure 6.5.6.5, and the WNN model also did well, which can be seen in Figure 6.5.6.4. Both of these models are not common, but performed admirably, beating the BasicCNN 2, which would have been a common choice in most scenarios like this. Our understanding of the machine learning models is limited despite designing them as during the training. It is hard to easily understand all the intricacies at

play. Models like Vis can help with this, but as seen in Figure 6.5.6.2, there are compromises to understanding what is going on more easily. All this goes to show that these models are very situational and that better understanding the math can help pick models more accurately but not perfectly.

Part VII: Discussion and Conclusion:

Different starting models result in different distances to an optimal model and varying computational costs per situation, resulting in the fastest starting model being extremely situational. This can be seen in both the experiments and past research through the fact that each model operates differently and performs wildly different depending on the situation during testing. The Vis 2 model is a good example of this with it being extremely successful in only one situation, but failing in most others. The experiments also showed that much of the performance can be estimated by comparing the input data to the mathematics behind the models. This is to be expected but is often underutilized due to some of the challenges when it comes to exploring the math behind these models and visualizing high-dimensional data. The structure at the start of training is extremely important and deserves more attention before training starts in most situations, as how a model performs can be predicted by understanding the math.

Starting models are incredibly important to how fast and computationally efficient each model performs. This paper examines how the math might connect to how different starting models perform given different situations, but only in a limited fashion. There is much more extensive research in activation function selection than in the rest of the starting model. This makes sense as activation functions are easy to switch and have a large impact, as shown in section 6.4, but other factors need more research. For now, all that can be concluded is that

starting models are important and that many of the factors of model situational-based performance can be extrapolated from the math.

While there is no one model that performs the best or an easy method of choosing a starting model, there are several methods that can be applied to help choose starting models. In this paper, it has been shown that training behavior is connected to the neural network algorithms, as the data being trained on. While there has not been much research on how this can be applied, visualizing the data can provide some information to assist with model selection. While the data often has too many dimensions to be fully visualize, looking at portions of the data can help to identify characteristics, such as whether the data is periodic, linear, or is simply extremely chaotic. This information alone does not give enough information to pick the best model for the situation, but it can help remove models that are unlikely to learn. One potential addition to this examination is short-term testing, where several models are trained for a few minutes or hours. While there is no easy way to determine which will perform the best, running each of these models for a short period and comparing their training rates and speeds can give a general idea of how different models might perform during long-term training, given specific situations. There is no easy way to identify models, but there are tests and general guidelines that can be followed to improve training speed. However, more research is needed into making an easy system for identifying proper starting models. Proper starting models will make machine learning more efficient as it spreads across the world, as well as making it more convenient.

While machine learning models are spreading rapidly, the knowledge of how to properly select starting models is still lacking in many areas. There are just so many compounding factors that make studying this difficult, not even considering the math or the situational aspects. While there is still a lack of resources currently, it is being studied and will likely result in improved

efficiency in the near future. This improvement in efficiency will also allow more people to run large models, allowing for more experimentation and an exponential increase in information about starting models and machine learning.

People active in the field of computer science, who work with neural networks and similar algorithms, understand that there is no model that will work every time. Normally, in the process of creating a model, you test different activation functions, neuron counts, and several other factors until reaching a model that is adequate. While it is understood that there is no best model for every situation, there is still a lot of guesswork. Resources for proper starting model selection are improving, but this issue is relevant and will not disappear, as it is not an easy task. We need to move forward into an age where we do not have to guess about what model is the best for each situation, but that appears to be a long way off.

APPENDIX:

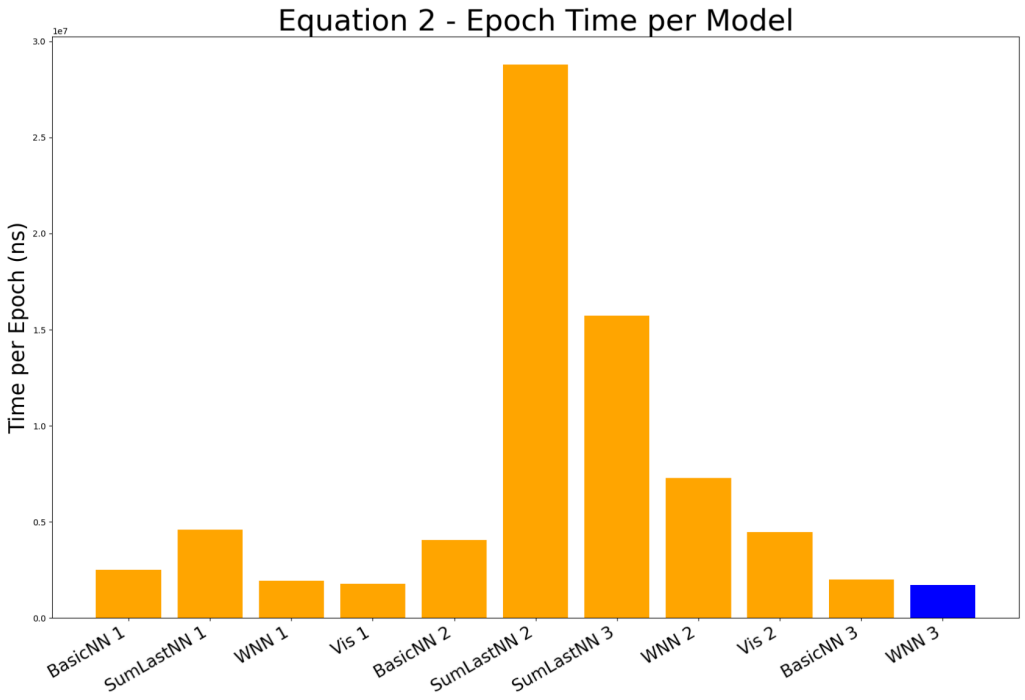


Figure 6.4.2.1 Shows which models performed the fastest computations per epoch (in blue)

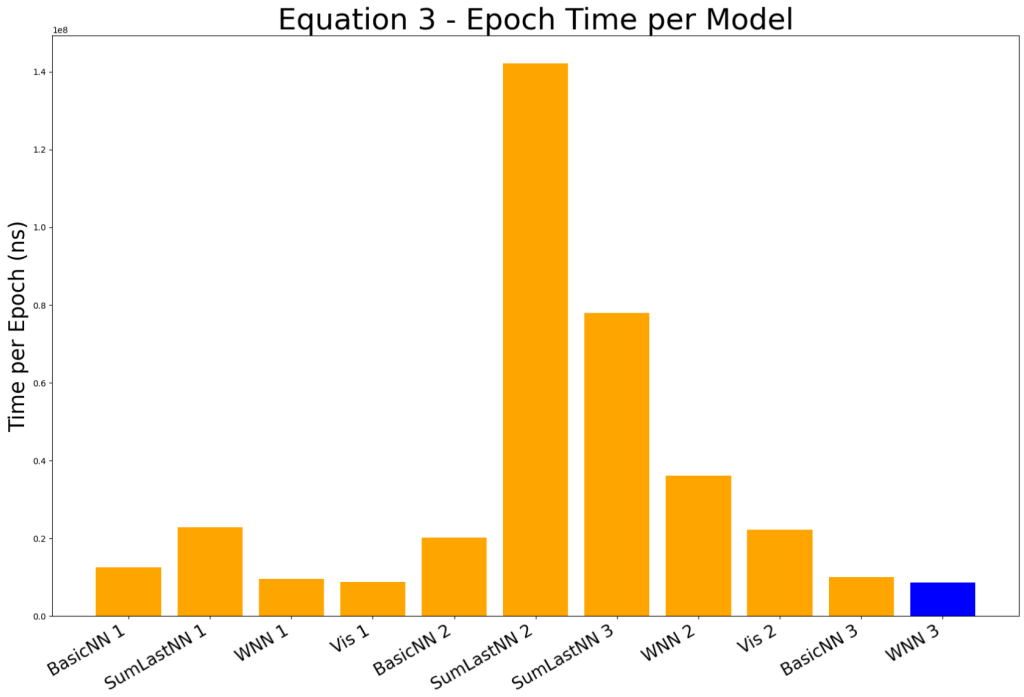


Figure 6.4.3.1 Shows which models performed the fastest computations per epoch

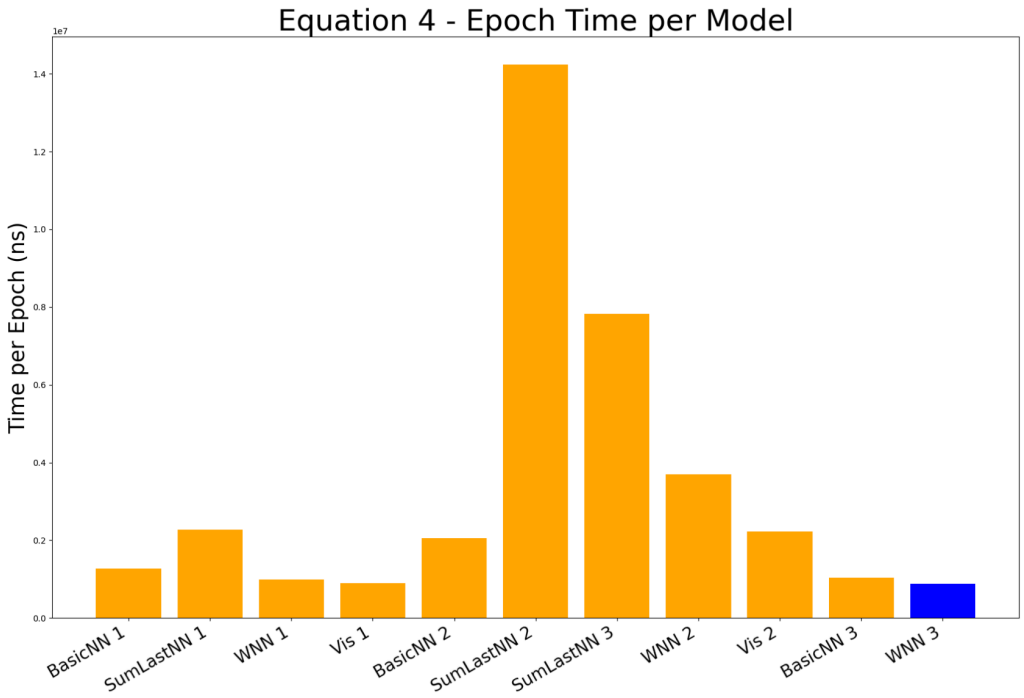


Figure 6.4.4.1 Shows which models performed the fastest computations per epoch

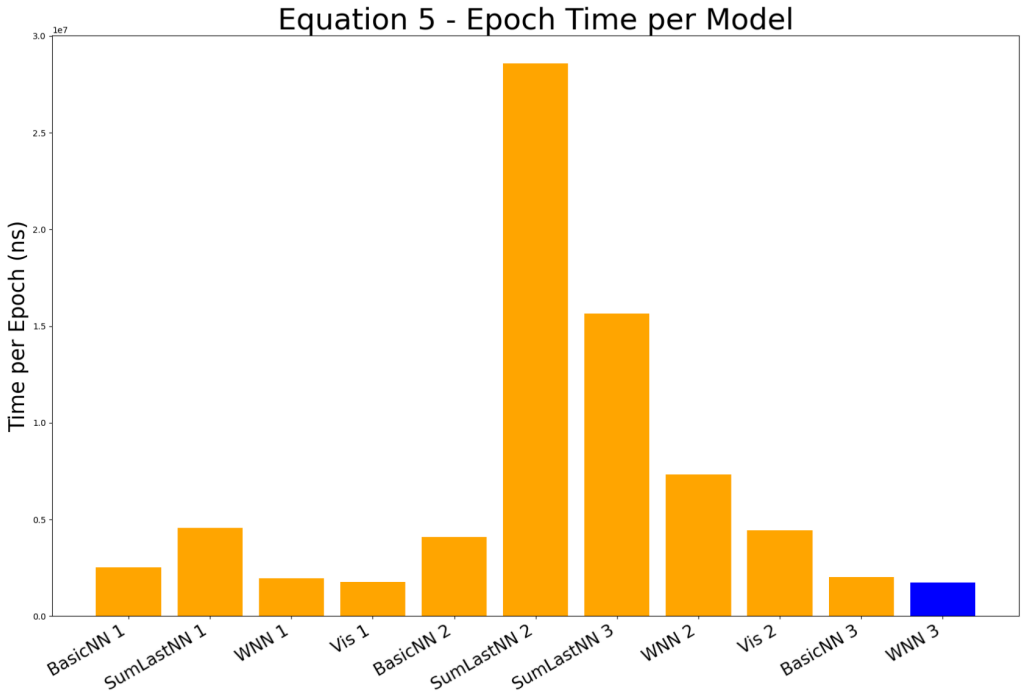


Figure 6.4.5.1 Shows which models performed the fastest computations per epoch

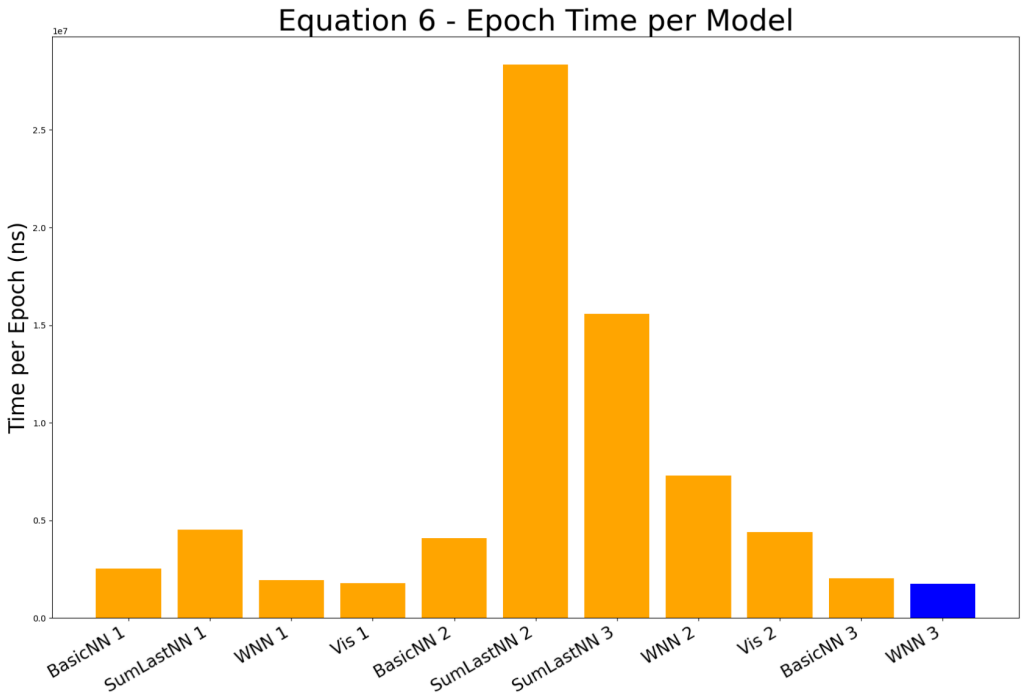


Figure 6.4.6.1 Shows which models performed the fastest computations per epoch.

Works Cited

Bilski, Jarosław, et al. "Fast computational approach to the Levenberg-Marquardt algorithm for training feedforward neural networks." *Journal of Artificial Intelligence and Soft Computing Research* 13.2 (2023): 45-61.

Chadha, Gavneet Singh, and Andreas Schwung. "Learning the non-linearity in convolutional neural networks." *arXiv preprint arXiv:1905.12337* (2019).

Dato-on, Dariel. “MNIST in CSV.” *Www.kaggle.com*, 2018, www.kaggle.com/datasets/oddrationale/mnist-in-csv.

Horak, Dean S. “Spiking Neural Networks: The next “Big Thing” in AI?” *Medium*, Medium, 22 Feb. 2024, medium.com/@deanshorak/spiking-neural-networks-the-next-big-thing-in-ai-efe3310709b0. Accessed 16 Jan. 2025.

Laudani, Antonino, et al. "On training efficiency and computational costs of a feed forward neural network: A review." *Computational intelligence and neuroscience* 2015.1 (2015): 818243.

Metta, Carlo, et al. "Increasing biases can be more efficient than increasing weights." *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2024.

Myszewski, Dave, et al. “Neural Networks.” *Stanford.edu*, 2000, cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html. Accessed 13 Jan. 2025.

Nguyen, Derrick, and Bernard Widrow. "Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights." *1990 IJCNN international joint conference on neural networks*. IEEE, 1990.

Sanderson, Grant. “But What Is a Neural Network? | Deep Learning, Chapter 1.” *Www.youtube.com*, 3blue1brown, 5 Oct. 2017, www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi. Accessed 13 Aug. 2024.

Sapkal, Ashwini, and U. V. Kulkarni. "Modified backpropagation with added white Gaussian noise in weighted sum for convergence improvement." *Procedia computer science* 143 (2018): 309-316.

Swirszcz, Grzegorz, Wojciech Marian Czarnecki, and Razvan Pascanu. "Local minima in training of neural networks." *arXiv preprint arXiv:1611.06310* (2016).

Thimm, Georg, and Emile Fiesler. "Neural network initialization." *From Natural to Artificial Neural Computation: International Workshop on Artificial Neural Networks Malaga-Torremolinos, Spain, June 7–9, 1995 Proceedings 3*. Springer Berlin Heidelberg, 1995.

Wolf, Mark. “Enabling Quantum Computing with AI.” *NVIDIA Technical Blog*, 12 May 2024, developer.nvidia.com/blog/enabling-quantum-computing-with-ai/.

Zodhya. “How Much Energy Does ChatGPT Consume?” *Medium*, 5 July 2023, medium.com/@zodhyatech/how-much-energy-does-chatgpt-consume-4cba1a7aef85.