*Cover page template for Diploma Thesis*

Bulgarian Diploma Thesis

**The Construction a Series of Low-Dimension Neural Networks and a Comparison of Different Activation Functions between their Gradient Learning Curves**

Marin Gjolena, 200069500

**Student:** Marin Gjolena **, Date:** January 24th, 2022

*signature*

Supervisor: , Date:

*Signature*

Department of Computer Science, AUBG Blagoevgrad, 2022

Declaration template for Senior Project and Diploma Thesis

**Title**: The Construction a Series of Low-Dimension Neural Networks and a Comparison of Different Activation Functions between their Gradient Learning Curves

**Author**: Marin Gjolena

**Repository**: https://github.com/rin183461/Senior-Thesis-NN

**Abstract:** Beginning in 1958 at the Cornell Aeronautical Laboratory, the fundamentals of Neural Networks and consequential anointing advancements to Machine Learning were invented by American psychologist Frank Rosenblatt. After nearly a century, the field has grown amass in abstraction, complexity and development, advancing the discipline into the highly practical and reliable instruments of data processing that have found massive use in day-to-day use. Though, with said abstraction and introduction of many Machine Learning libraries in the public domain, it is not abnormal for one to utilize this knowledge without remotely understanding the interwoven harmony of algorithmic, mathematical and biological beauty the precepts of Neural Networks function through. This project aims to reexplore the origins of this highly important Machine Learning subfield by creating a program that is able to build, train and validate data, all from scratch using the Python programming language. This includes the implementation of forward and backpropagation, two algorithms in demand of adept understanding of the mathematics discipline, specifically in the network’s Linearly Algebraic nature of its input, output and hidden layers and the Calculus-revolved gradient vector of partial derivatives in the latter algorithm, through which the network will be trained. With the inclusion of modern optimizing algorithms, the network may be improved into a generalized tool of categorical analysis via the installment of the Categorical Cross-Entropy Loss function. The interlacing of all aforementioned elements then yields a model capable of producing dependable results. Furthermore, to highlight the staple of the field’s technological advancements, the project also sees to draw a cross-comparison between the now-widely used Rectified Linear Unit (ReLU) and the archaic Sigmoid activation functions. This aims to highlight their training efficacy and difficulty through both, a mathematical and empirical lens. All in all, this project is to be deemed as an academic resource touching upon the hidden knowledge and principles of Neural Networks and the progress of the field which have led to its massive success and use in today’s world.

Keywords: Neural Network, Project, Python, Training, Algorithm, Activation Function, Mathematics, Calculus, Linear Algebra, Forward Propagation, Backpropagation, Optimization, Comparison, Sigmoid, Rectified Linear Unit

**Declaration of authorship:**

“The Senior Project/Bulgarian Diploma Thesis presented here is the work of the author solely, without any external help, under the supervision of ….. All sources, used in development, are cited in the text and in the Reference section.”

Author: Marin Gjolena

# Table of Contents

[0. Table of Contents 5](#_heading=h.3dy6vkm)

[1. Introduction 6](#_heading=h.1t3h5sf)

[2. Theoretical Design of the Software Solution 8](#_heading=h.4d34og8)

[2.1 Layers 13](#_heading=h.2s8eyo1)

[2.2 Sigmoid Activation Function 17](#_heading=h.17dp8vu)

[2.3 Rectified Linear Unit Activation Function 18](#_heading=h.3rdcrjn)

[2.4 Softmax Activation Function 20](#_heading=h.26in1rg)

[2.5 The Loss Function (Categorical Cross-Entropy) 21](#_heading=h.lnxbz9)

[2.5.1 Accuracy 23](#_heading=h.35nkun2)

[2.6 Backpropagation 24](#_heading=h.1ksv4uv)

[2.6.1 Derivatives 24](#_heading=h.44sinio)

[2.6.2 The Chain Rule 28](#_heading=h.2jxsxqh)

[2.6.3 Derivatives of the Activation Functions 29](#_heading=h.z337ya)

[2.6.4 Backpropagation Example 31](#_heading=h.3j2qqm3)

[2.7 Optimization 34](#_heading=h.1y810tw)

[2.7.1 Adam Optimizer 36](#_heading=h.4i7ojhp)

[2.7.2 L1 L2 Regularization 38](#_heading=h.2xcytpi)

[2.7.3 Dropout 40](#_heading=h.1ci93xb)

[3. Design of the Software Solution 43](#_heading=h.3whwml4)

[3.1 The Functional Requirements 43](#_heading=h.2bn6wsx)

[3.2 The Non-Functional Requirements 45](#_heading=h.3as4poj)

[4. Implementation of the Software Solution 47](#_heading=h.1pxezwc)

[4.1 Libraries Used 47](#_heading=h.49x2ik5)

[4.2 The “Layer” Class 48](#_heading=h.2p2csry)

[4.3 Activation Functions’ Classes 49](#_heading=h.147n2zr)

[4.3.1 The “ReLU” Class 49](#_heading=h.3o7alnk)

[4.3.2 The “Sigmoid” Class 49](#_heading=h.23ckvvd)

[4.3.3 The “Softmax” Class 50](#_heading=h.ihv636)

[4.4 The “Loss” Class 50](#_heading=h.32hioqz)

[4.4.1 The “CategoricalCrossEntropyLoss” Class 51](#_heading=h.1hmsyys)

[4.4.2 The “Softmax\_Loss\_Aggregate” Class 51](#_heading=h.41mghml)

[4.5 The “Accuracy” Class 52](#_heading=h.2grqrue)

[4.5.1 The ”Accuracy\_Categorical Class” 52](#_heading=h.vx1227)

[4.6 The “Optimizer\_Adam” Class 52](#_heading=h.3fwokq0)

[4.7 The “Dropout” Class 53](#_heading=h.1v1yuxt)

[4.8 The “Input\_Layer” Class 53](#_heading=h.4f1mdlm)

[4.9 The “Model” Class 54](#_heading=h.2u6wntf)

[4.10 Visualization 56](#_heading=h.19c6y18)

[4.11 Upgradeability 57](#_heading=h.3tbugp1)

[5. Testing 58](#_heading=h.28h4qwu)

[6. References and Supplementary Material 62](#_heading=h.nmf14n)

# Introduction

This project aims to build a program that explores two problems through two fronts, both mathematically and empirically: the construction of a set of neural networks capable of fitting a prediction on a spiral scatter, and the best approaches in initializing and maturing the network in the most optimal way possible.

The technologies through which I aim to approach this project are few. Firstly, the chosen programming language is Python, as it is a powerful and simple language to work with. The former quality is effective to me, insofar as the constraints of the project allow. For this reason, I only aim to extract tools that comply with the specifications of the Senior Project, and as such, at the time of writing, the only library that I aim to work with is the NumPy library. As the research progresses, I may find myself adding to the list of technologies to assist me in the project.

The NumPy library is a math-focused toolbox. The field of Neural Networks is one whose backbone heavily relies in the simplification of gargantuan quantities. A neural network is, after all, man’s attempt in understanding the very instrument of understanding (the mind, that is). As the name itself suggests, the network mimics the behavior of organic neurons, and as the demand of the network’s output becomes more complex, so do the neurons that populate it. Abstracting the vast number of parameters that the network needs to function is necessary in being able to digest and manage the sheer volume of the knowledge at hand. Thus, Linear Algebra becomes a necessary discipline to dilute the numbers’ volume. The NumPy library specializes in introducing Linear Algebra to Python programming, hence its inclusion in the development of this project.

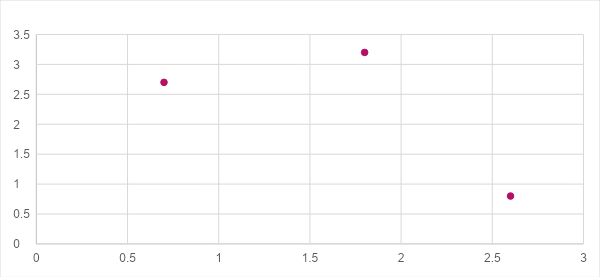
Machine Learning and Neural Networks are fascinating fields in their goal to animate an otherwise static machine. For man, the brain has been the throne of thought and intellect and by extension, humanity’s overall evolutionary success. A success that relies on this organ that evolved to feel a hunger only satiable by true intellectual brilliance. It is through the ML field that one bestows this sense of the mind (albeit fragmented) unto an object bereft of one altogether. This analysis and process are precisely what lie in the heart of my inspiration in pursuing this project. Lastly, there is one more objective I aim to fulfill as I write this thesis. The reason I can appreciate the beauty of this project is mainly through years of understanding the demanding fields of Computer Science, Multivariate Calculus and Linear Algebra. However, these fields are inaccessible to an average reader equipped with only a modest background in all mentioned fields. This knowledge is locked behind barriers of complicated dogma that constrain said reader from the veiled elegance and charm Machine Learning has introduced. As such, it is my own personal goal that I write a theoretical approach that is accessible for reading not only by the select few that undergo the study of advanced curricula, but also by those with minimal academic background fueled by curiosity to, at the very least, develop an intuition of the complex beauty of Neural Networks.

And one more thing, this study is just that, a study. I may, and will, make a phrasing mistake here or a calculation mistake there. But I qualify that the first objective of a senior thesis is to, first and foremost, learn and prove that one can learn. Learning is a process of mistakes and improvements, not unlike what the very topic of this project covers.

# Theoretical Design of the Software Solution

It is in this section that I discuss how the program has been designed and the explanations of each modification in the program. In order to explain how parameters affect neural networks, it is vital that the objective of a neural network and its parameters are explained first.

A Neural Network (NN for short) has a very simple task at its core: to fit a curve through some data. When we train an NN we focus on stretching, squishing, flipping and shifting a curve to fit a data set.



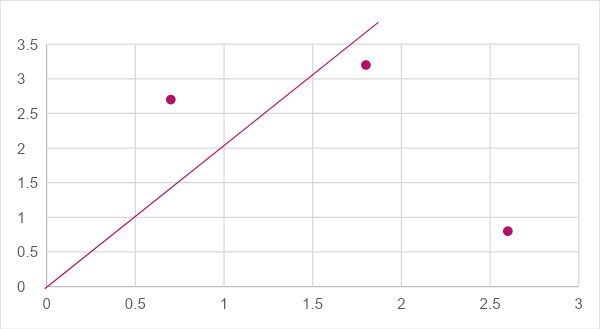
*Figure 1: Curve fitted to some data*

Certainly, you may say that you can easily draw a curve through some points of data, but this is true for only a very small number of dimensions. How would you go about drawing a curve through four dimensions? What about when the layers span hundreds of dimensions? This is the reason NNs are so powerful; the qualitatively-demanding aspect in which a human would find immense difficulty navigating is not so much so for a computer. This is the reason NNs are such a sought-after topic in the IT world.

Back to the construction of networks, an NN fundamentally consists of just one element: the **neuron**—a product of inputs and parameters called Weights and the Bias. Firstly, a neuron may easily be perceived as simply the product () of summed Weight-input multiplications () and the Bias ():

It is the intrinsic non-linearity, or “bend” that we can manipulate through linear operations (stretch, squeeze, rotate and shift) to produce a curve that fits our data. And this “bend” (non-linear, “not like a line”) is achieved through what we call **Activation Functions**. Examples of activation functions include the Sigmoid, Rectified Linear Unit or ReLU, SoftPlus expressed as

respectively. There are many other such functions and one important property that they all share is **non-linearity**. Non-linearity is important because it depicts that the function’s graph is “bent,” an attribute the network cannot achieve otherwise since every other operation (achieved through the other two elements, the weights and biases) is linear. Hence, if we want to fit a line to some data, the line with which we work with should have some form of intrinsic bend that we may modify to fit a dataset.



*Figure 2: There cannot be a single straight line that can be bent into a shape such that it fits the data, nor can one single line accurately measure the data*

As mentioned, we require that the activation function through which the data passes through as it travels to a neuron must be fit to the data, for we cannot possibly hope that the elementary form of an activation function will fit the data directly. This is where the next two parameters of an NN: the weights and the biases come into play.

In very basic terms, what a weight is responsible for is the stretch and orientation of the curve. Not unlike a basic mathematical linear equation, when the slope is modified, so is the steepness and orientation of the line. Let us take these two examples:

𝑓(𝑥) = 2𝑥

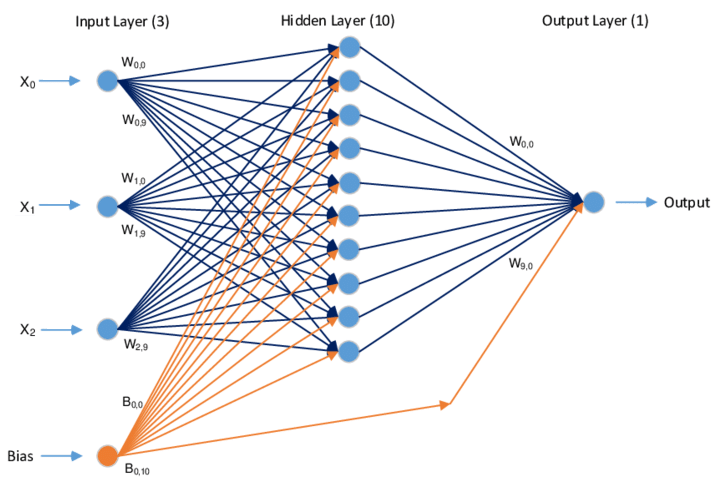
𝑔(𝑥) = 0.5𝑥

We know that if we observe the graphs of these two functions, we get two lines, one (𝑓) steeper than the other (𝑔). As such, for a given interval of values we receive a stretched and squished set of solutions (e.g. for 𝑥 = 2, 𝑓(2) = 4 whereas 𝑔(2) = 1). Furthermore, adding a negative sign to these functions flips them over, hence rotating them.

And lastly, the bias, which mathematically is just the curve’s up or down shift. For example:

It is clear that 𝑔 is one unit higher than 𝑓. When the NN generates a curve that may fit the data, it will have to be shifted in place, just like how a puzzle piece needs to be shifted in the correct spot somewhere where it fits in a jigsaw.

But how do all these elements come together? Like so:



*Figure 3: Simplified view of an NN (adapted from Tekerek)*

Note: The most common visualization of biases is to imagine an additional neuron solely responsible for the biases of a layer. However, I tend to imagine the image more simply: every neuron (except the input layer’s) bearing its own unique bias.

This is an NN in the very, very simplest of depictions. And there is another interpretation to the way the NN functions as a whole through these parameters and that is the more classical understanding: that the weights and biases determine which neurons fire and which ones do not, like a very complex logical circuit where the weights and biases serve as logical gates which determine how the data flows. How the curve that the NN generates is trained though is a totally different kind of beast.

Hopefully these analogical facets have shed some light in how the fundamental innerworkings of an NN function.

## Layers

The numerous algebraic computations of neurons within the layers contribute to the generation of a complex structure, capable of fulfilling a (complicated) task. The outputs of a layer can be ordered in a mathematical object known as a **vector**. So the best thing to start with is the implementation of the layers in the solution – in the form of the input, output and hidden layers, that is. Since these collections of data are multidimensional, it would be best to use Python lists which connotate with vectors and matrices in Linear Algebra. As you may imagine, in order to feed the inputs forward from one layer to another, all the inputs from one layer must be multiplied by all the weights in the next layer. If we have, say, a very simple network consisting of two neurons in the input layer, three neurons in the hidden layer and two neurons in the output layer, the total number of weights between

the input and hidden layers is six: two weights for every neuron in the hidden layer (three) that correspond to the two input neurons. Let us have a notation for these neurons so we may orientate ourselves more easily and less arbitrarily. Michael Nielsen, an expert in the deep learning and quantum computing fields, uses this notation in his *Neural Networks and Deep Learning* online book to refer to weights in an NN:

such that:

This way, we avoid unambiguous references to different weights in the network. The weights: are both multiplied by the inputs of the input layer in the first neuron of the hidden layer like so:

The bias notation is similar:

such that (just like the weights):

Every neuron in the hidden layer undergoes the dot product operation. I.e. the first neuron in the hidden layer receives two inputs that are coupled to two weights, and then are summed. That is precisely how the dot product operates. There is one rule that the dot product demands though, and it concerns the dimensions of a two-dimensional matrix. The dot product can only take place if the column dimension of one matrix matches the row dimension of the other. I.e.:

is invalid. Though if it so happens that there is a mismatch between the two dimensions, one can use transposition to swap the dimensions: rows become columns and columns become rows:

The column dimensions match, but this operation remains invalid up until we transpose:

This operation is now possible.

But what exactly do the weights do? As previously mentioned, an NN aims to fit a curve through data. Weights are coefficients that modify the shape of a curve. Weights squish, stretch and rotate the curve. Multiplication by coefficients comply with linear transformations.

After using the dot product, we will need to add the bias to the operation. This is easily done and understood, since every neuron holds a bias, all we do is add a bias to the dot products:

What about the biases, what do they do? If weights are responsible for shaping the curve, the biases work to shift the curve up or down. While the curve of some function may be able to fit the data, it must be shifted into position such that it fits said data.

Lastly, the final thing to do is to envelop these operations in the activation function of choice, denoted with σ:

More generally, the activation () of the j-th neuron in the l-th layer, given an arbitrary of inputs and weights, looks something like so:

This entire set of operations connotate with the operations that the neural network processes in the hidden layer.

What happens for this one layer expands for the entirety of the network. This is the feedforward portion of the network that takes in inputs and runs them throughout to produce an output.

The input layer is “special” for it bears no biases. How inputs are passed is also very important in generalizing the network so it can get used to the variance of inputs it receives after training. Not providing variance in batches atrophies the network from variable combinations of inputs. Hence, it becomes practical to train the network in **batches**. Batches contain a list of valid input lists, otherwise known as **samples** or **feature sets**. Taking in many samples at a time is beneficial for the NN. Furthermore, training the network in batches is much faster as it makes use of parallel processing via the many cores GPUs (Graphical Processing Unit) in our computers have.

## Sigmoid Activation Function

The Sigmoid function is the original one to be used in the procession of the hidden layers back when neural networks became more integral. The Sigmoid was a goo choice, for its outputs spanned between zero and one for all possible values, which means that the outputs are a form of “certainty.” The closer to one, the more certain the output is, and that is legible by us humans as we are familiar with the concept of percentages, which derive from one such interval. The formula for the Sigmoid is:

However, due to its training demand, this function was later replaced by the Rectified Linear Unit activation function. However, it will be used in this project to demonstrate its efficiency and training speed in comparison to the ReLU activation function. As the values reach either or , the tangent lines flatten, which, for reasons to be explained later in this paper, will become problematic when it comes to training.

Chart, line chart

Description automatically generated

*Figure 4: Sigmoid Function Graph (Wood)*

## Rectified Linear Unit Activation Function

The ReLU function is much simpler than the Sigmoid, which means that calculations become much simpler (and less risky, because there is no exponentiation and potential overflow errors). The formula is very simple:

All this means is that when , and when , . The graph makes this all the clearer:

Chart, line chart

Description automatically generated

*Figure 5: ReLU as compared to another activation function, the Softplus activation function (Stowell)*

Because we are simply checking whether the output is positive or negative, and simply keeping it the same or equalizing the input to , it is already pretty clear how much simpler than the Sigmoid function computation becomes for the program. Also, unlike the Sigmoid, ReLU does not plateau, which means that its tangents are much easier to compute and hence training becomes much faster.

One could ask, well is this not a linear function? It almost is! It is a composition of two linear functions, but it is not linear. Look at the break at . That is all that is needed for non-linearity, it is not straight! The almost-linear attribute of its being makes training easier, and its nonlinearity ensures we can create complex shapes. Truly a paradoxical marvel in such a simple function.

## Softmax Activation Function

The NN I am studying will have to solve a classification problem – that is, a scatter of spirals into well-defined classes. What helps in this instance is the **Softmax** activation function:

This function is typically used in the output layer to transform the outputs which may be numbers of different magnitudes into more meaningful, readable data. Harrison Kingsley and Daniel Kukieła, two credible ML scientists, in their *Neural Networks from Scratch in Python* explain the function in a series of simple steps, the first one be **exponentiation**. The outputs are all put through a very familiar function in the Math field:

𝑦 = 𝑒𝑥

where 𝑥 in the Softmax formula is replaced with 𝑧𝑖,𝑗, where 𝑖 = 𝑐𝑢𝑟𝑟𝑒𝑛𝑡 𝑠𝑎𝑚𝑝𝑙𝑒 and 𝑗 = 𝑐𝑢𝑟𝑟𝑒𝑛𝑡 𝑜𝑢𝑡𝑝𝑢𝑡 𝑜𝑓 𝑡ℎ𝑖𝑠 𝑠𝑎𝑚𝑝𝑙𝑒. There is one problem exponentiation raises though, and that is “exploding values.” By exploding values we refer to values too large for the program to run correctly, and the Softmax function is a potential source of exploding values. To avoid this potential issue, we utilize two mathematical properties of Softmax:

Using these two properties, we can offset the values (codomain) to a more manageable interval of numbers while maintaining their distributions through normalization, which will be discussed in a moment. We do so by subtracting the maximum value in a sample: i.e the highest value from a sample’s inputs is subtracted from the entire sample. After subtraction, the largest number will then be 0, and the lowest, some negative number. Thus, we isolate the values from getting too large for the variables not to crash the program.

After having exponentiated and accounted for their potentially large values, the outputs are put through what is called **normalization**. Normalization is what scales down the exponentiated outputs to a distribution of probabilities that a human can easily understand, from 0 to 1. This tells us why the predicted output is what it is in the NN’s eyes. These are what we call **confidence scores**: the confidence of the network in producing its outputs. This is what the denominator’s summation aims to achieve. When confidence scores are produced, they give us an idea of how good or bad the network is performing through a clearer point of view. For example, if an output produces these values:

Confidence Scores Indices

Then the output with the highest confidence is the third one, which has a 50% confidence rate. The output the network would choose is the one it is the most confident in choosing, so the third entry would be its choice. To note the index of the output of its choosing, this list goes through a function called **Argmax** which checks all confidence scores and returns the index of the highest one. In this example above, it would return 2 (keep in mind that indices in programming usually start from 0). Argmax becomes particularly useful when taking in a batch of samples, since for each sample, we would know the network’s chosen output since we would have a list of indices that relate to the network’s outputs with the highest confidence.

## The Loss Function (Categorical Cross-Entropy)

This marks the long and short of the feedforward part of the network. The true underlying complexity lies in when one trains it. Training a neural network is an intense and voluminous process, as is the nature of anything dabbling in Linear Algebra. In order to tell a network how to improve, it has to know how wrong it is first. This is achieved through what is known as a **Cost**, **Error** or **Loss** function. A Loss function is an evaluation of how deviated a predicted result (by the network) is from the observed (true) result. A good measure of error is the difference between these values:

where denotes sample loss value, is the natural logarithm, is the i-th sample in the set, is the output index (label) as noted by Kinsley and Kukieła. However, we have to keep in mind that the neural network in our focus aims to categorize data. A very special function was thus created, specifically to handle this form of operation. The function is called **Categorical Cross-Entropy**:

where 𝐿𝑖 denotes sample loss value, log is the natural logarithm, 𝑖 is the i-th sample in the set, 𝑗 is the output index (label), as noted by Kinsley and Kukieła. Though daunting in sight, we introduce a quality-of-life method that simplifies our work considerably.

Since out of 𝑛 possible outputs only one is correct, we may label this fact via something called **one-hot encoding**. The idea of one-hot encoding is to generate a vector where one of its values is ”hot” by a value of 1, while all the other entries are zero. This one- hot entry bears the index of the correct (predicted) value of a sample and is given the value 1 in the output vector.

Since every other entry is zero, the sum in the categorical cross-entropy formula becomes redundant since all entries but one are multiplied by zeroes from the one-hot vector and the only non-zero entry is 1, and any number multiplied by 1 is the number itself. Thus, the whole formula simplifies to:

where 𝑘 is an index of the target label (ground-true label).

### Accuracy

Another instrument with which loss is accompanied is **accuracy**, “which describes how often the largest confidence is the correct class in terms of a fraction” (Kinsley and Kukieła, ch.6, p.24). Using the values – that is, the observed (true) values – and the Argmax function output which labels the network’s highest confidence index from one-hot encoding, we may cross-compare the predicted and observed values in a batch of samples. Whenever the predicted sample outputs un/equal that of the observed samples’, we store that information in a new list in the form of ones (equal) and zeroes (unequal). Then, we take the mean of all the values in this new list and what is generated is the accuracy. For example:

Network Softmax Outputs Confidence Scores Observed Values

Comparing the two batches, the list we get is:

The mean of which is:

So, the example values we took let us know that this example NN has a chance of producing reliable outputs.

## Backpropagation

Up until now we have observed how a neural network processes an input to produce a prediction (output), and a manner of tools through which we can check how right or wrong these predictions are. But how do we use this information? Certainly, a measure of error tells us how wrong the neural network is, but it does not tell us which parameters are faulty and which ones are not. This is not unlike receiving a bad test score without being told without being told where the mistakes are. Criticism is meaningless if it is not constructive.

Fortunately, there exists a way to determine the impact each parameter has upon the generation of outputs. The method through which we explore the improvement of the network’s parameters is the so-called (and so-dreaded) **backpropagation**. This is a relatively Multivariate Calculus-heavy algorithm, dependent upon the knowledge of derivatives and their meaning in the context of neural networks. I will only touch upon the vital information necessary for the understanding of the algorithm, for the connotations derivatives bear are extremely broad and complex (and truly magnificent). There are many resources dedicated to a more detailed explanation of derivatives and multivariate calculus.

### Derivatives

Let us begin with what derivatives are. You may recall Newtonian Laws of Motion in your Physics class, where some of the most important concepts revolved around Position (an object’s location at a given moment in time), Velocity (an object’s rate of change with respect to its position) and Acceleration (an object’s rate of change with respect to its velocity). However, what you may have not been told is that these three concepts are mathematically connected. Velocity is the derivative of position, and acceleration is the derivative of velocity. Let us focus on just the latter duo.

Velocity is what we associate with speed (though speed in Physics is a bit different from velocity [ ]), and speed is a concept we understand well. It is a measure of how fast we are moving in space. Acceleration is a bit more abstract but still something we easily perceive. You may think of it as a measure of how fast your speed is changing, i.e. how quickly your speedometer’s needle is moving. You can also feel acceleration at work in the form of inertia, whenever your velocity is changing (accelerating/decelerating), you feel like something is pushing you (towards the opposite direction of movement). This is why safety belts are so important: in case of sudden stops, velocity comes to a halt near instantaneously, which means that deceleration goes to a considerably high rate, the resulting inertia of which may become hazardous in its severity, hence the usage of safety belts.

We can observe the loss of a neural network in a similar context. We would like an NN’s errors to be as minimal as possible; more explicitly: the closer to zero the loss function, the better. But we do not know which parameters affect the network in the way it does. Randomly modifying the weights and biases in hopes of achieving a satisfactory result is very inefficient in the long run. Thus, we observe the derivatives of all the network’s parameters and how they are affecting loss in the same way acceleration affects velocity. Knowing how every weight and bias contributes to loss lets the NN know how to nudge all its weights and biases in such a way that it minimizes errors in the future.

Hopefully, this description has served in forming an intuition around derivatives. In Mathematics, **derivatives** are described as **the rate of change of a function with respect to a variable**. There are many other definitions of a derivative, but even if the concept remains opaque, it is such a researched subject that its background mechanics have been simplified into a set of abstracted rules which eludes the necessity of a well-established understanding. However, the knowledge behind the fundamentals of Calculus is certainly fascinating and well-worth any time invested in it.

Let us now proceed concretely, starting with what the **partial derivative** is. The partial derivative measures the effect a single variable has on a function’s value and is denoted using the operator like so:

The partial derivative is a consequence of calculating derivatives of functions that have multiple variables. In our case, these variables are all the weights and biases of an NN. Recall that a derivative is the rate of change with respect to a variable. When a function has multiple variables, we focus on deriving just with respect to one and disregard the others. For example, if we have a function of three variables , where are the input variables, then the derivative of with respect to would be:

Taking all possible partial derivatives with respect to all variables of :

Finding the partial derivatives with respect to all of a function’s variables turns out to result in the creation of a very important mathematical object in multivariate Calculus. The **Gradient** is a vector composed of all partial derivatives of a function. Think of the partial derivatives as fragments or parts (hence *partial* derivative) that make up a whole (the gradient). The gradient is denoted by the nabla symbol: . Using the previous example:

So how are derivatives calculated? Well, there are a few rules. Skipping all background, here is a small list of derivative rules we will need. For visual simplicity, one alternative notation for derivatives uses an apostrophe to imply the derivative of the function (another form of notation uses the letter “d” to differentiate derivatives from partial derivatives: i.e .:

Note the fourth rule, when constant multiply the variable with which the function is in respect to, it is not affected.

Let us take an example:

How about derivatives with multiple variables? We simply ignore the other variables and treat them as constants (recall the four rules mentioned above):

While we are at it, we can compose the gradient of :

### The Chain Rule

There is one more rule to include, and this is easily the most important rule regarding derivatives in respect to this thesis: the **chain rule**. One thing to keep in mind is that a neural network is nothing but a really big function composed of smaller functions. As such, even the most atomic property (such as the weights and biases) are part of a larger chain that composes the entire network. The chain rule allows us to traverse the chain and reach the derivatives of these variables despite the complexity of the function they are nested in. Let us take an example with two functions: and :

where is the variable of , and is substituted with . We then pass as the input variable of . is then substituted with . Clearly, they are all somehow linked, but the connection between and has become a bit abstract. This whole thing can be rewritten as:

which makes the relationship a bit clearer. is the input of which is the input of . So then, how do we produce the derivative of with respect to ?

This is where the chain rule comes in, “which tells us that the derivative of a function chain is a product of derivatives of all the functions in this chain” (Kinsley and Kukieła ch.8). Referring to the previous example:

This is but a glimpse of the mayhem the chain rule will unravel when we follow the humongous chain of functions that the Loss function is host to. But it the one tool that will allow us to intelligently improve our NN.

Now, let us proceed with the derivatives of the functions we will use in our program. I will not go into detail as to how these derivatives are produced, though some of these derivatives are simple to understand and short on words.

### Derivatives of the Activation Functions

Let us start with the easiest: ReLU’s. As we know, the ReLU function is simply . This can be rewritten as:

To calculate the derivatives, we simply take the derivatives of both functions for their respective domains. The derivative of is as we know, and is a constant so its derivative is just :

The partial derivative simply focuses on the variable we are respecting, for example:

where simply means “ is if , otherwise it is 0.”

Moving on to the Softmax partial derivative. Recalling the formula of the Softmax function:

where denotes the j-th Softmax’s output of the i-th sample and is the input array as a product of the previous layer. To keep things simple, we will simply take the derivative pre-computed since its generation is beyond the scope of this paper:

where , known as the Kronecker delta function, is:

The implementation of this is derivative is challenging but not undoable. However, in a moment, we will see that there exists a neat solution to the derivatives of the Softmax and Categorical Cross-Entropy formulae when they’re multiplied, with which we can easily compute our values.

Next, the loss function’s derivative we will take it to be:

When we implement, we will be aggregating these two latter derivatives into their simplified product. All in all, the solution of the partial derivative of the loss function with respect to the inputs of the Softmax, which per the Chain Rule is:

Simplifies to just:

This solution is much simpler, and it gives us direct access to the inputs of the Softmax function with a simple subtraction of the predicted and observed values. It is now that we have all the pieces to our backpropagation and put them together.

### Backpropagation Example

Let us take an example. Say we have this network from the *Neural Networks from Scratch in Python* book:

Diagram

Description automatically generated

*Figure 6: ReLU activation applied to the neuron output (Kinsley and Kukieła, ch.9)*

The Softmax and Loss derivatives are excluded, but let us assume the derived values, which we will call **dvalues** from the aggregate derivative () is . The inputs are , the weights and the bias . Let us begin with backpropagation:

First we take the derivative of the ReLU function. Since the value is , then the derivative should be :

So multiplying by the next layer’s derived value using the chain rule we have:

We proceed with the inputs of the ReLU function, the sum of the bias with the weight-input dot product. There are four variables that are linked to the sum. Recall the chain rule order:

The partial derivative of the sum operation is always 1, with the variable of respect and inputs having no impact:

So the aforementioned list’s values are all equal to 1:

We have the bias’ derivative, now the last thing to go through is the chain of the multiplications , and with respect to the weights and inputs. Using the partial derivative’s multiplication rule:

Now we have all the partial derivatives of all the parameters of the example, and this wraps up backpropagation. We would simply have to add the Loss and Softmax derivatives in that order from some Loss value to complete the picture, but that is more well understood when the simpler remainder of the network’s backpropagation process is exemplified, as above.

## Optimization

Now we move on to training the network using the information yielded from backpropagation known as **Gradient Descent**. More specifically, we will be looking at **Stochastic** Gradient Descent or **SGD** for short. Let us see what that is and how it works.

But first, it is worth mentioning what the name means (and its confusing background). As such, I cite here Kukieła and Kinsley (ch.10, p.7):

The first name, Stochastic Gradient Descent, historically refers to an optimizer that fits a single sample at a time. The second optimizer, Batch Gradient Descent, is an optimizer used to fit a whole dataset at once. The last optimizer, Mini-batch Gradient Descent, is used to fit slices of a dataset, which we’d call batches in our context. The naming convention can be confusing here for multiple reasons.

First, in the context of deep learning and this book, we call slices of data batches, where,

historically, the term to refer to slices of data in the context of Stochastic Gradient Descent was mini-batches. In our context, it does not matter if the batch contains a single sample, a slice of the dataset, or the full dataset — as a batch of the data. Additionally, with the current code, we are fitting the full dataset; following this naming convention, we would use Batch Gradient Descent. In a future chapter, we’ll introduce data slices, or batches, so we should start by using the Mini-batch Gradient Descent optimizer. That said, current naming trends and conventions with Stochastic Gradient Descent in use with deep learning today have merged and normalized all of these variants, to the point where we think of the Stochastic Gradient Descent optimizer as one that assumes a batch of data, whether that batch happens to be a single sample, every sample in a dataset, or some subset of the full dataset at a time.

All in all, despite the batch size, what happens is practically the same: the parameter gradients that are generated by backpropagation are multiplied by a small, positive factor called the **learning rate** (usually denoted as ) to compute a new value denoted as , the vector of changes.

This new value is then **subtracted** from the parameters so that we guide them towards the direction of **steepest descent**, in other words, towards a **minimum**.

Though this is the general approach to lowering the value of a Loss function, it is worth mentioning that it is practically guaranteed that the Loss function will exhibit more than one minimum. These minima are called **local** minima. The one we ideally seek is the **global** minimum, the lowest minimum of them all. While SGD is the correct approach, it is too barren to optimally seek the global minimum. It is possible that for small learning rates the gradients are stuck in a local minimum and cannot get out, not unlike a ball trying to get out of a trench with inadequate speed. One may think that tweaking the learning rate so that it is a bigger number might help, but that may simply make the gradients “explode,” and thus cause the parameters to go flying all over. Thus, to avoid these problems, SGD optimizers are enhanced with clever tricks through which we overcome these difficulties.

Additionally, NNs can also suffer from something called **overfitting**. *Overfitting and Undercomputing in Machine Learning*, and article by Oregon State University professor, Tom Dietterich, explains that overfitting is the risk the learning algorithm’s attempt goes through in figuring out the best fit to some data during training. Said risk involves the fitting and memorization of the **noise** in the training data. Think of data as a desert and noise as a particular sand dome; the terrain and domes of the desert churn and shift continuously, so memorizing a dome as part of a desert is useless. It is much simpler to think of the desert as a big dome in and of its own, **generalizing** the image. When the NN is overfitting, it is trying to fit its predictions to an annexed view of the whole image, thus causing inaccurate predictions.

There are many such methods to bettering the optimizer and avoiding these problems. The ones I have used here are three:

-Adam (short for Adaptive Moment Estimation) Optimizer

-L1 L2 Regularization

-Dropout

I will go over each one and explain what they are and how they work for the optimizer of my program.

### Adam Optimizer

The Adam optimizer makes use of a concept called **momentum** in tandem with **adaptive**, **per-weight** learning rates and a **bias correction mechanism**. Let us begin with what the concept of momentum is. Imagine momentum as the inertia in the physical world: if a ball is thrown uphill, depending on how big or small the force applied was, the inertia it gains will determine whether it reaches the top and rolls over it. Momentum attempts to do the same for parameters that risk getting stuck in local minima. Not only that, in the case that the gradient wobbles around a global minimum without stability, momentum’s information about the previous informs the gradient towards the best direction to head towards

Momentum generates a rolling average of gradients over a number of parameter updates and is utilized by the next gradients to be generated at each step. It is then used to set a parameter in an interval from to that is used to depict the fraction of the previous parameter’s influence on the pass and subtracting the current gradient that is being multiplied by the learning rate.

We also introduce the idea of **learning rate decay**, which decreases learning rate during training, as the name would suggest. There are several ways of doing this, but our plan is to decay on a per-step schedule. This is also referred to as **1/t** or **exponential decaying**. The goal is to update the learning rate on every step of the training loop using the reciprocal of the step count fraction, which is the learning rate decay hyperparameter. The further in training, the larger the step is and so is the result of this multiplication. The reciprocal is then taken and multiplied by the initial leaning rate. An added ensures that the product of the algorithm never increases the learning rate.

Up until now, we have taken in consideration only one, globally-shared learning rate. Adam and other optimizers abandon this strategy and opt to use a per-parameter learning rate, meaning that every parameter has its own . By storing previous updates to parameters, we can control how much or little the parameters change. For example, some weights tend to become much bigger than others, and that is a problem for generalization since the network may rely too much on a portion of neurons while others “die.” This set of changes is stored in the corresponding parameter (weight or bias) **caches**. Furthermore, the bias correction mechanism is applied not just to the caches, but also to the momenta! Yes, we are including the momenta here too, correcting them to compensate for the initializations which start at zero, before they warm up with initial steps. The correction is done via dividing both momenta and caches by . As step rises, , whose initial value is , tends towards 0 in a gradual manner:

The same applies to a similar hyperparameter , for which the starting value is and also approaches 1. It is by these values that the momenta and caches are divided by, correspondingly. Since these are fractions, dividing by them increases values and hence training speed in the starting stages before both tables warm up during initial steps (Kinsley and Kukieła, ch.10, p.63). The calculations are thus:

The parameters are then adjusted like so:

With this, we end our brief overview of the Adam Optimizer. Adam is the most popular and widely used optimizer for the reasons stated above, and can help the network escape local minima, be stable and reach the global minimum more smoothly, which is why I chose to implement it in the program. It is worth noting that Adam is a sort of amalgam of other optimizers, built atop another optimizer called RMSProp (Root Mean Square Propagation) and borrowing from the SGD with momenta and AdaGrad (Adaptive Gradient) algorithms, which is probably why it has achieved the greatest amount of success. But this does not necessarily mean it is the best for a problem and that other optimizers may be able to perform better in certain cases.

(epsilon) is a hyperparameter that simply is added to the denominator to avoid division by .

### L1 L2 Regularization

Let us move on to L1 and L2 Regularization. Regularization aims to reduce errors in generalization, which we earlier discussed. L1 and L2 regularization are used to generate a **penalty** which is added to the Loss function’s value in order to toll a model for weight and bias values that might be getting out of hand. Big-valued weights may be an indication of the network learning by heart and becoming inflexible towards exotic scenarios, which is why they need to be reined in. Shortly put, L1 regularization is the sum of all the absolute values for the weights and biases. L2 regularization is the sum of the squared weights and biases—one is linear, the other not so. L1 punishes small weights mostly, whereas L2 targets larger weights and biases. Both L1 and L2 regularizations force the network to use its whole computational power to generate predictions. A value referred to as **lambda** is added to the equation to scale the penalty: the higher it is the more influential it becomes.

L1 weight and bias regularization:

L2 weight and bias regularization:

And to the loss equation is added:

As regularization is added to the overall loss, which means it has an overall influence on it, then it also goes through backpropagation. Fortunately, their computation is pretty early on in the process and quite simple since these are just absolute value and power functions.

Derivatives of the regularization:

This marks the end of regularization.

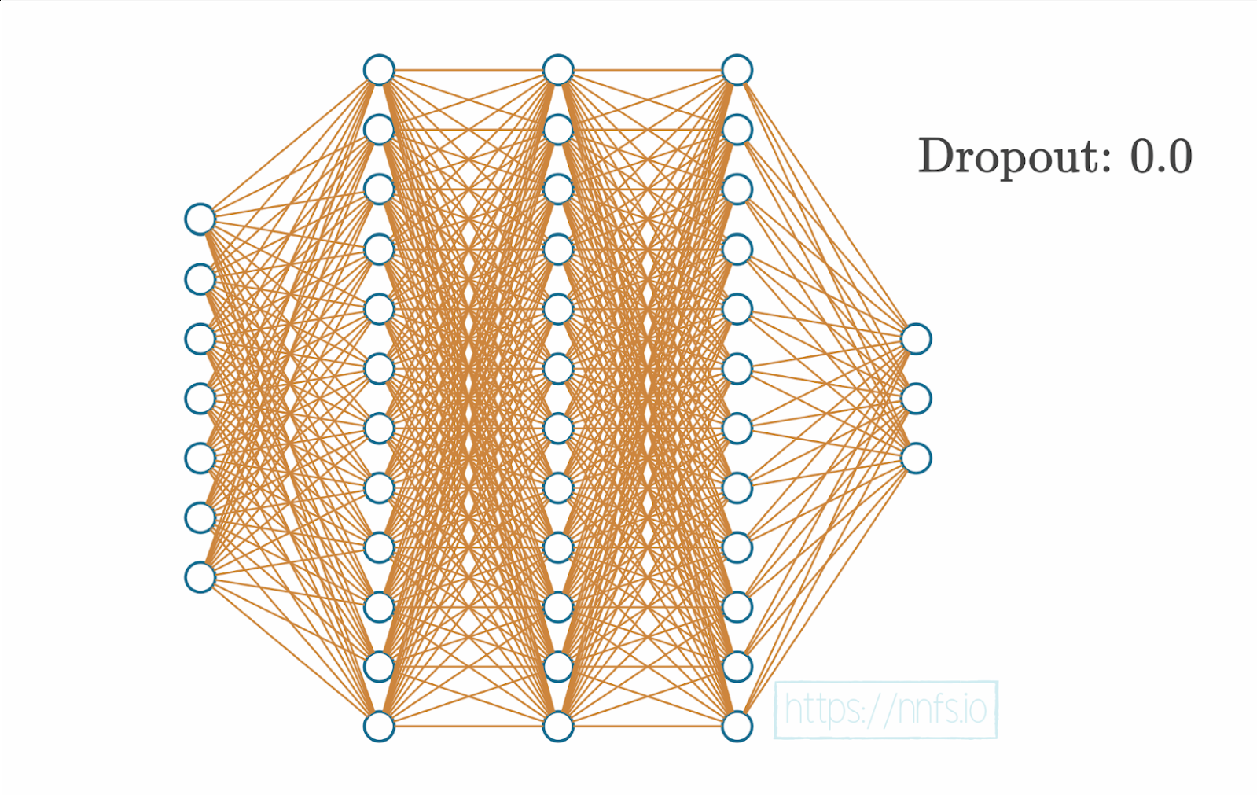
### Dropout

It is time for the last method of optimization: dropout. Dropout is a rather simple concept: it refers to the random shutdown of neurons in the network for a forward pass **during training**. This enables the neurons that remain for that forward pass to be trained and the randomness ensures that over many such forward passes, all neurons will statistically shut down to allow others to be trained. This helps greatly with generalization and prevents the network from relying on specific neurons and not the entire network.

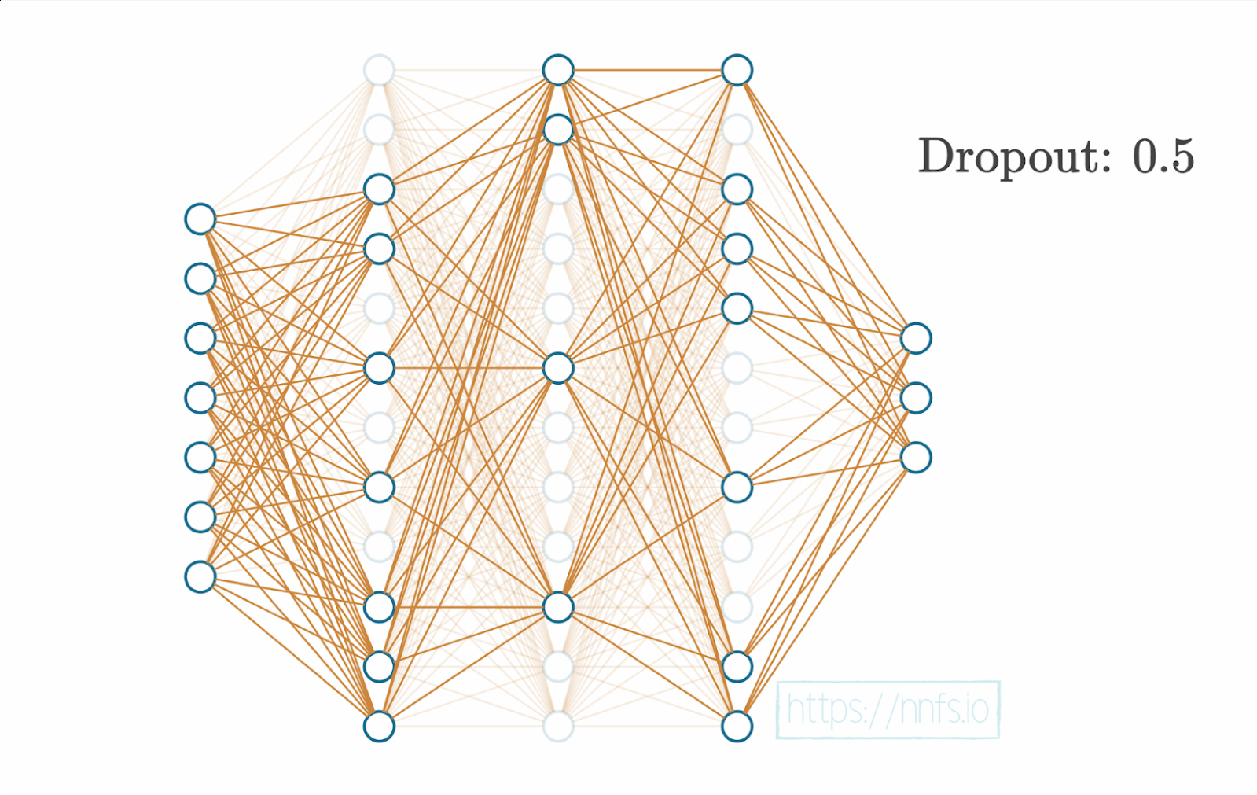
Another problem it helps with is **co-adoption**, which is where neurons in a layer depend on the preceding ones to compute the values without grasping or learning the underlying function on their own. Additionally, dropout also helps with data noise, which we mentioned before.

The way the network will decide to “turn off” neurons during training will be done through what is called a **Bernoulli distribution**. Bernoulli distributions are the “Booleans” of probability, so to speak, where a value of is given at probability and a value of at probability. If we have some random value, say, , then:

Below are some pictures that display dropout visually (Kinsley and Kukieła, ch.15, p.8):



*Figure 7: Dropout of 0%*



*Figure 8: Dropout of 50%*

As for backpropagation and its derivative, it is somewhat similar to ReLU’s. Let us denote Dropout as :

where denotes the index of the given input and the layer output (Kinsley and Kukieła, ch.15, p.14).

This marks the end of optimization methods and the overall innerworkings of a neural network.

Since the project aims to demonstrate a neural network at work, data preprocessing has not been a priority, but it is a thing of importance in the training of real-life objects, especially when data is scarce. But since this work aims to only build, train and evaluate NN models, with the material covered already being quite abundant, this thesis’ theoretical implementation will stop here.

# Design of the Software Solution

In designing the software solution, an OOP-focused (Object-Oriented Programing) approach is often the best approach to follow, especially when the program can discretely be segmented into manageable pieces that we can ascribe classes to. The classes are designed in such a way that they are cohesive when they are aggregated but not too coupled to each other to preserve maintainability.

This approach has also allowed me to upgrade the program in a direct way (especially when implementing Dropout) without having to change other classes or be afraid of anticipating change in the program’s behavior. This will be further discussed later in this section.

However, firstly I will begin by introducing the functional and non-functional requirements that I had in mind when implementing the project.

## The Functional Requirements

Here is a list of all the program’s functional requirements:

**Declaration of a number of inputs, as per the user’s needs**

The user should be able to pass as a valid input a number of variables. This number should accept a number of inputs as per the user’s needs. This is tied to the number of neurons in the first, or input layer, which is achieved through a list type.

**The addition of (hidden) layers, as per the user’s needs**

The user should be able to add as many hidden layers as they may need to as far as network complexity goes. The declaration of these layers extends the first functional requirement, where the number of neurons per layer are also determined by the user and are not limited by any limitations (other than Python’s). This requirement also involves the declaration of the applicable number of weights and biases the layer requires for the procession of inputs and the production of outputs. The same mechanic that is responsible for the hidden layers should function for the output (last) layer, as there is virtually no difference between the nature of these layers.

**The addition of activation functions through which the layers are filtered through**

An important feature the program must have, is the inclusion of different activation functions which will affect the resulting outputs of the layers. This is an extremely important feature as it ties to one of the main objectives of the program – cross-comparison between the activation functions’ performance during training, that is.

**The connection of the layers to form a cohesive model**

The program should be able to connect the declared layers into a cohesive model from which the following aspects of the build can be easily controlled (training and evaluation).

**Forward propagation, and hence evaluation or processing, of the inputs into a valid output**

The program should be able to automatically process the passed data into a valid output (valid in the sense of mathematical operations, not in respect to its trainability). This requirement should also be automatically invoked during training given a number of iterations (epochs).

**The network’s capability to be trained**

The network should be able to be trained given a number of iterations (epochs) through which it can improve its predictive capabilities and accuracy. The features should involve the implementation of the backpropagation algorithm and the inclusion of a valid optimizer. The method, other than its declaration, should be automatic and be able to train the network to the best of its capability without mathematical errors.

**Visualization of the models’ predictions via graphical and written data for cross-comparison**

The network should log the training data (achieved accuracy and minimization of loss) on the console and be able to, by the end of training and evaluation, produce readable graphical data for the network’s prediction pattern, given the data set.

## The Non-Functional Requirements

**Including the NumPy Library for the fastest possible computation speed**

The NumPy library features well-optimized methods for the Linear Algebra (and other) processes that are required for the implementation of the Neural Network theory.

**Ensure generalization through the use of L1 L2 Regularization**

L1 L2 Regularization is an important concept that regulates not just the theoretical stability of the models the program is tasked to build, but also the integrity of the program itself. The concern of exploding values, discussed in §2.5.2, also concerns overflowing value errors. This error occurs whenever a variable is unable to store values beyond its capacity. In our case, Python’s floating-point variables are fixed to 32 bits, and as such, cannot hold anything higher in value than or

and from what I have seen, Windows does not permit any alterations to these variables to 64-bit or higher. For further information, consult §2.5.2

**Ensure optimal training by implementing the Adam optimizer**

The Adam (Adaptive Momentum) Optimizer is statistically the most widely used optimizer with the highest chance of success in training, both in speed and network generalization. For more information, consult §2.5.1.

**Include Dropout to further prevent model overfitting**

In order to generate a reliable model, overfitting must be discouraged during training. Including an optional Dropout feature should alleviate the problem to a significant degree. For further insight, consult §2.5.3

**Aggregate classes whenever applicable for faster computation (this ties to the algebra of the Softmax and Cross-Entropy Loss derivatives)**

According to Kinsley and Kukieła, an aggregate class that combines the Softmax and Categorical Cross-Entropy classes in the implementation can increase computation speed by up to seven times (ch.9 p.66). This ties to a mathematical simplicity concerning the product of the derivatives of the Softmax activation function and the Categorical Cross-Entropy formulae.

**Ensure that the addition (and removal) of layers in the network is easy**

By making use of the OOP paradigm, adding and removing layers from the network should be simple and intuitive by simply initializing user-defined variables through a Model class that unifies all the different aspects of Neural Networks together.

# Implementation of the Software Solution

What comes with the theory must not be put into practice. The code and implementation of the project has been rather thoroughly commented within the program itself, so I would only be repeating what has been already written out within the code. This section will simply showcase how the program’s control flow functions and what the program in general is doing during execution.

It should be of no surprise that the entire solution will be dealing intensively with lists (or otherwise known as arrays), which play the role of the mathematical objects known as vectors and matrices. All issues regarding Linear Algebra have been already discussed in §2 and will be referring to the theory discussed there for the implementation and will not focus on reiterating what has already been said.

The whole project also aims to follow the Object-Oriented Programming paradigms in the form of classes for each of the separate problems which will then be neatly aggregated into a unified class with which full control will be granted from a concentrated object. The code’s architecture was achieved through Kinsley and Kukieła’s *Neural Networks from Scratch in Python*.

## Libraries Used

Several libraries are used for the creation of this project:

* NumPy – Used for algebraic operations such as the dot product and other math-related methods
* NNFS – A library supported by Kinsley and Kukieła, it simply provides abstracted methods from the Matplotlib library to supply the spiral data scatters and the random initialization of parameters.
* Matplotlib – Used for data visualization of the data
* Time – Used to measure execution time during training

## The “Layer” Class

The “Layer” class is responsible for the creation of a number of lists which will hold the weights, biases and number of neurons for the layer and the number of inputs from the previous layer. This is done through the initializer:

def \_\_init\_\_(self, n\_inputs, n\_neurons,

l1\_weight\_strength=0, l2\_weight\_strength=0,

l1\_bias\_strength=0, l2\_bias\_strength=0):

self.weights = 0.0001 \* np.random.randn(n\_inputs, n\_neurons)

self.biases = np.zeros((1, n\_neurons))

Note the *l1\_weight\_strength*, *l2\_weight\_strength*, *l1\_bias\_strength* and *l2\_bias\_strength* parameters. These are the parameters responsible for setting L1 L2 regularization for the layer.

self.l1\_weight\_strength = l1\_weight\_strength

self.l2\_weight\_strength = l2\_weight\_strength

self.l1\_bias\_strength = l1\_bias\_strength

self.l2\_bias\_strength = l2\_bias\_strength

Now we define the *forprop()* method which stands for forward propagation. This name will be used to forward propagate all other classes in the implementation. This method invokes the dot product of the weights and the inputs and then adds the biases of the neurons in the layer to produce the output list.

def forprop(self, inputs, training):

self.inputs = inputs

self.output = np.dot(inputs, self.weights) + self.biases

Note the *training* parameter. This is used to differentiate between training passes and validation passes, very useful for when dropout is implemented so that the network knows when to apply it or not.

Then we define the *backprop()* method which stands for backpropagation. This is the method that takes in the derived values of the next layer in the form of the *dvalues* parameter and outputs the derivatives of the current layer. The way the derivatives are computed follow the rules discussed in §2 for multiplications, sums and regularizations, and multiplies them by “dvalues” as per the Chain Rule.

Note the *axis* parameter in the *self.biases* member. The axis allows us to do operations only with respect to one dimension, in this case, the rows’ which are the biases. We will be using *axis=1* more often, which connotates to working with columns.

## Activation Functions’ Classes

There are three activations functions that the project studies: ReLU, Sigmoid and Softmax. The Softmax function is mutually shared between all models and is to encompasses the output layer of every network, whereas the other two are to be used in separate models in the hidden layers for the cross-comparison study.

### The “ReLU” Class

The “ReLU” class is not difficult to implement. The *forprop()* method saves the inputs and returns positive inputs as they are, and negative inputs as 0. The *backprop()* method copies the derived inputs from the next layer and returns 0 if the input was negative, or 1 if the input was positive (§2.2). The *predictions()* method simply returns the *outputs* parameter.

### The “Sigmoid” Class

Similar to the ReLU method, we are simply abiding by the formulae of the Sigmoid (§2.3) and its derivative and implement accordingly using the *forprop()*, *backprop()* and *predictions()* variants.

### The “Softmax” Class

The Softmax function class follows the same blueprint as the aforementioned activation function methods in architecture, but the *backprop()* method is removed and will be melded in a separate class with the Categorical Cross-Entropy derivative for faster computation. For further detail, consult §2.4, §2.6.3 and §3.2.

As discussed in §2.4, the exponentiated values are clipped by the maximum value during calculations to avoid overflow errors.

## The “Loss” Class

The Loss function class will serve as an abstract class. It is responsible for the calculation of loss and regularization loss that is generated in the “Layer” class (§4.1). Whenever the L1 L2 parameters of the “Layer” object is valid (greater than ), regularization loss is calculated (using the appropriate formulae, §2.5.2) and returned like so:

Notice the *self.trainable\_layers* iterator in the loop. The trainable layers refer to the layers that are, well, trainable. These layers are received through this method in the class:

def store\_trainable\_layers(self, trainable\_layers):

self.trainable\_layers = trainable\_layers

Trainable layers are those that have weights (or biases, one list is enough). This will be made use of to discern the input layer from all the others (activation functions included) later in the Model class.

Now we also need a way to calculate average loss per batch of inputs to form an idea about how well the model is predicting values during training. Since this is a general calculation, regardless of what function is used to calculate loss, we include it in this class (since we will be inheriting from it later).

The *output* argument takes in the model’s output, *y* is the observed value from the training data and then the bare asterisk. The bare asterisk is used to force the caller to use named arguments, so *include\_regularization*. Note that the *sample\_losses* variable is using the *forprop()* method. This is making use of dynamic binding with the inherited classes, which is why this action is not illegal and actually very handy and elegant.

### The “CategoricalCrossEntropyLoss” Class

The Categorical Cross-Entropy Loss class will be a child of the “Loss” class and inherit the methods through which it will be able to calculate average loss and regularization penalties. As always, we include a *forprop()* method (consult §2.5 for the formula), but no *backprop()* (as per §4.2.3).

Similar to the Softmax function, as discussed in §2.4, the Loss function’s values are clipped to prevent overflow. The conditional statements check whether the passed labels are one-hot encoded or sparse (the numbers contained are the correct class numbers). The conditional statement checks the dimensions of the passed parameter, if the dimension is , the values are sparse. If they are two-dimensional, then there’s a set of one-hot encoded vectors. For the second case, by multiplying confidences by the targets, zeroing out all values except the ones in the correct indices along the column axis (*axis=1*).

### The “Softmax\_Loss\_Aggregate” Class

Here we have the separate aggregate class of the Softmax and Loss functions, which holds only the *backprop()* function. I attempted to unify the classes together using objects for easier management through this class, but it only made things more difficult and increased coupling unnecessarily. This solution became more elegant for backpropagation, in the end.

Instead of performing the subtraction , we can make use of the being *y\_true*, which means that we are accessing one-hot encoded vectors. This means there is only one value of in these lists and the remaining elements are all zeroes. This allows use to use NumPy to index the prediction array with the sample number and the true value index, taking off from the values. In case the conversion to true labels is not done, we add additional code to handle the conversion in case the shape is not the true labels’.

## The “Accuracy” Class

The accuracy class, not unlike the loss class, will function as an abstract class and be the parent of future children classes to emphasize on upgradeability. This class calculates accuracy by comparing the predicted values with the target labels and computes their average. This *calculate()* method exists only in the children classes and makes use of dynamic binding. Then the accuracy is returned.

### The ”Accuracy\_Categorical Class”

The “Accuracy\_Categorical” class is tasked with calculating accuracy for the Categorical Cross-Entropy formula. Other loss functions which may be implemented in the future will, just like this class, inherit from the abstract parent class “Accuracy” from which the mean will be calculated. If the values are not one-hot encoded, the *compare()* method makes a check and returns the encoded list which is then handled by the *calculate()* method in the parent class.

## The “Optimizer\_Adam” Class

This class is one of the more quantitative, but it really is just a bunch of formulae (from §2.5.1). Before any calculations take place, a *pre\_update\_parameters()* method is added. This simply checks whether the *self.decay* parameter is anything but and if so, to add it in calculations. The formulae are separated into four lists, two for the momenta and two for the caches. After some algebra, they are added to the parameters. Then, similar to the *pre\_update\_parameters()* method, a *post\_update\_parameters()* method is implemented. This simply adds to the iterator tracking.

## The “Dropout” Class

The dropout class aims to implement the effects of Dropout (§2.5.3) by passing a “rate” hyperparameter by which we induce the percentage of the layer that is “shut down” (Bernoulli distribution). A *forprop()* method is invoked which, when the “training” Boolean is set to false, is not invoked and simply returns the inputs as outputs, not affecting the result of a model during evaluation. Otherwise, the method makes use of the NumPy library to zero out the output of the neurons using the *numpy.random.binomial()* method.

This method includes three parameters: *n* (number of experiments), *p* (probability of the true value of the experiment) and *size* (array size produced). Setting these parameters to *n=1*, *p=self.rate* and *size=inputs.shape* generates the appropriate binomial distribution that mimics the outputs of the layer type it precedes, but modifying them by the rate of neurons to disable passed in declaration. The division by the *self.rate* variable is done to simplify the derivative of the dropout function during backpropagation and is removed by multiplying it for the outputs of the *forprop()* method. This value is saved to avoid computational redundancy.

## The “Input\_Layer” Class

The Input layer class exists conceptually to make sense of the Model class, as we are going to be passing in a list of objects and will require this class in order to pass in the inputs. It will only have one method that simply returns its passed inputs as outputs.

## The “Model” Class

The Model class is the unifier of all the aforementioned classes that allows for simpler creation, management and training of the neural network models a user may want to construct. An initializer sets the loss, optimizer and accuracy objects automatically, while declaring an empty list where the layers will be stored in. An *add()* method is added to the class in order to add a layer to this list.

The more complicated part of this code is the connection of the layers, via a *connect\_layers()* method. This method begins by initializing an input layer via the *Input\_Layer* class. Then the number of layers added to the list is counted to use as a maximum to a range later when we loop through the layer list. This number is stored in a *layer\_count* variable.

Another list is declared, this containing the trainable layers. As mentioned in §4.4, this is used to discern between non-trainable layers.

Then we add the layers added through the *add()* method in the *self.layers* list by appending them. Using Python’s built-in *prev()* and *next()* list methods, we refer to the elements of the list and modify them appropriately in the *self.layers* list in an ordered manner. We need to fix it by adding the first layer, that is, the input layer. Whenever *i==0* then the previous element must be the automatically declared input layer; this is the true “first” layer, but as mentioned in §4.8, this is a class that simply serves the formality of adhering to the object-revolved handling of the layers and is not declared by the user. The next element then must be whatever the 0-th element used to be before the *self.layers* list was modified. This continues so long as the iterator *i* remains lesser than the range of the loop.

When this condition breaks, then the last element cannot just be the layer that the user has added, we would be missing the Loss function, so it is appended to the *self.layers* list. Additionally, the output of the last activation function (Softmax) is also saved to be used for predictions later.

Furthermore, to train, we need to know which layers are trainable. Using the *hassattr()* method, the *self.layers* during iteration within the loop is checked in whether it possesses a *“weights”* member. If so, it is added to this list that makes it easy to train valid layers. We pass these layers to the Loss class’ method *store\_trainable\_layers()* that we discussed in §4.4.

After the loop is done modifying the lists appropriately, we end the *connect\_layers()* method by declaring an object from the aggregate class *Softmax\_Loss\_Aggregate* for later use.

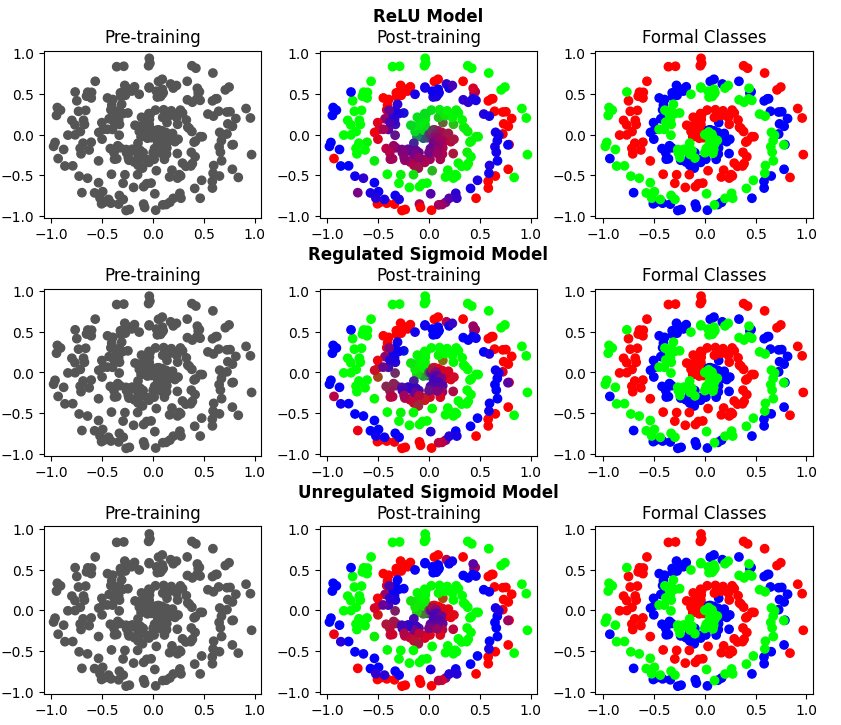
We add, in good practice to the rest of the implementation, a *forprop()* method accepting the list of inputs and the *training* Boolean to discern training from evaluation and decide whether to de/activate the dropout layer. Then the Softmax layer is returned as the last layer in the *self.layers* list.

This output is passed on to the *backprop()* method of the class, again, in coherence to the rest of the program’s architecture. Firstly, we call the *backprop()* method from the aggregate Softmax Cross-Entropy class and update the layers’ *dinputs* variable, which passes in the derived inputs of the next layer to the current one (recall that backpropagation happens in reverse). Then the layer list goes through a loop which accesses all its elements in reverse using the *reverse()* method. All elements are accessed except the last one, since that is the input layer which bears no backpropagation.

After all is done, we then begin training the network using a *train()* method that accepts the inputs, observed outputs, number of epochs, printing frequency and validation data (should there be any, to discern training from evaluation). Within this method a loop that iterates epochs within the inputted range of epochs added . Outputs are stored in an *output* variable using the *forprop()* method. Data and regularization loss are generated by making use of the Loss class’ *calculate()* method and they are added to provide total loss. Afterwards, predictions and accuracy are generated and stored in like-named variables. Then the *backprop()* method is called, passing in the predicted and observed values, after which the optimizer is called and invoked for all trainable layers. Then the relevant information is printed on the console. If a *validation\_data* tuple parameter is passed, the network performs just a regular pass with no backpropagation or optimization and the *training* argument is passed as *False* so that the Dropout layer is deactivated.

## Visualization

Using the Matplotlib library, we define a method *create\_subtitle()* that divides a plot made of subplots into a number of rows. It receives figure, grid and title parameters. A *row* variable adds a subplot in the grid and sets a title. Then the subplot is hidden and only the title remains. Then, the results before and after training are stored in lists. Then a figure of nine subplots is created (since we have just three models) and plot the figure into three rows and columns. The grid is plotted and the rows are named appropriately. Then the axes are named and the predictions pre- and post-training, together with the formal classes are submitted to the graphs and then the following image is generated:



*Figure 9: Visualized Models’ Outputs*

## Upgradeability

For the sake of upgradeability, the Loss and Accuracy classes have been implemented in an abstracted way for easy implementation of any additional formulae in the corresponding categories.

A more comprehensive initializer function may be implemented to further simplify the creation of models via a list of parameters that can take in a number of layers, neurons per layer, activation function for the hidden layers and so on.

The visualization of the data is done in a hard-coded manner just for the demonstration of the models’ outputs. It could have been integrated within the model class, but for the purposes of this project it has been left out.

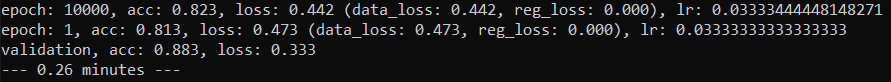
# Testing

Three models were instantiated for testing. These models were:

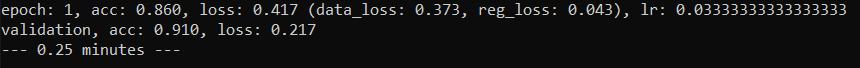
* A ReLU model
* A Sigmoid model with L2 regularization
* A Sigmoid model without regularization

These models aim to exhibit the different capabilities (and shortcomings) of Neural Networks of the chosen architectures and yield some form of conclusion. Each test will take in a scatter of three spiral classes and be tasked to classify them appropriately. Each model will be trained for 10,000 epochs and is built using just one hidden layer carrying 126 neurons. For obvious reasons, the input layer receives two inputs and the output layer predicts three classes, so it constitutes to three neurons.

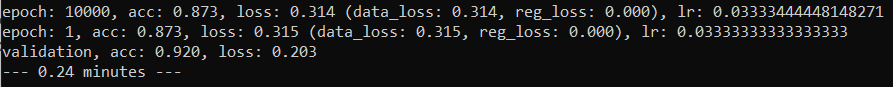
These are the performances of the models in order:



*Figure 10: ReLU Model Results*



*Figure 11: Regulated Sigmoi Model*



*Figure 12: Unregulated Sigmoid Model*

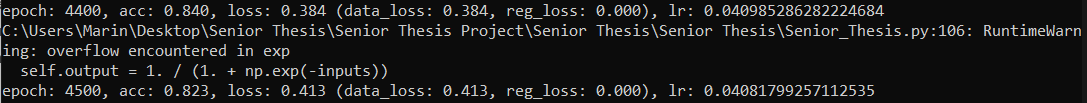
*epochs* depicts the number of epochs for which I have taken only the 10,000th, *acc* depicts accuracy in decimal percentage, *loss* is the estimation of the loss function (where its composition between data and regularization loss is highlighted), and *lr* depicts loss rate (how much loss the training is remedying). Lastly, an execution timer measured how long it took for these trainings and evaluation to compute in total.

The first model’s statistics are the most surprising since it seems to have performed the worst on both fronts: execution time and accuracy. I am not very certain as to what is causing this outcome since the code appears to be working well and the computations are simple, no matter how one looks at them. The only way I can explain this is that my hardware is may be playing a part in this since the laptop I am working on does have some anomalies of its own (which is unfortunate).

I have not run this code in any other machine, so I cannot say for sure. Mathematically, the ReLU model should have done the best, yet here we are. Another possible cause might be the optimizer. Perhaps a different optimizer such as AdaGrad or RMSProp may have performed better, and fortunately through the architecture of the software solution, upgradeability is very simple and integrable.

The second model performed slightly better than ReLU’s, sitting at a .25 minute execution time and an accuracy bump of over 3 percent. It is uncertain how it is achieving these numbers, however, there is a noticeable difference regarding the two models in the earlier epochs of their training. After the 100th epoch, the first model had an accuracy of around 60.7%, whereas the Sigmoid one around 39.3%, which means that the first model outperforms the second for scarce epochs of data by around twofold, which is a big positive for the ReLU model since not always can one procure large numbers of data (which is where data preprocessing comes into play).

Thirdly, we have the last model. The unregulated Sigmoid model allows the parameters to take any values with no limit, resulting in a risk of overflow errors. This model tends to fall victim to this around the 4000+ epoch mark:



*Figure 13: Third model overflow*

This in turn deactivates some neurons completely, allowing the network to rely upon a smaller number of neurons which hastens executions time, which is obviously not something we look for in our models. However, what is most surprising is that this model appears to have performed the best in terms of accuracy too, scoring an accuracy score of 92%. This, I assume, is the result of overfitting: since the number of neurons has decreased and the network is also being fed the entire data set as an input, the risk of overfitting is much higher due to its limited capacity, even with the inclusion of a dropout layer. It is very possible that the entire network is relying upon fitting its prediction pattern to fit the specific data set, never accounting for a shift in the spirals’ curvatures should a modified set be fed as an input.

Additionally, this model is, to all ends and purposes, kaput. Though the overflow error occurs around the 4000 epoch mark, it still means that the network’s risk of running these errors is too high and also that whatever is left of it is also probably falling victim to overfitting. This network may, eventually, break down completely.

Now let us have another look at figure 9 from §4.10:

Diagram, qr code

Description automatically generated

*Figure 14: Networks’ Predictions, visualized*

There is a lot “blending” occurring at the center of the first model’s predictions (instead of reds and blues, the predictions are purple, indicating overlapping). This is good news, I believe, for it means that the model has determined that spirals in general are convergent in the center and that they tend to overlap as they converge. Knowing this, the model accounts for the potential overlaps and becomes more forgiving in how it differentiates between the spirals at the center. As they fan out, the network becomes more poignant and specializes the spirals’ arms into their appropriate classes.

This is not the case for the latter two models. The second model does exhibit some blending akin to the ReLU model’s, but it is nowhere near as prevalent. Whether this model is doing a better job in classifying the spirals or not is up for debate, since it also appeared to perform better during training which means that its optimizer might be favoring the Sigmoid more than the ReLU model.

The third model is obviously doing the “best.” There is barely any blending and it has categorized the spirals almost perfectly. For the aforementioned reasons, this is probably just a bogus result and the predictions appear favorable only at first glance. This model is very probably a victim to overfitting.

Overall, the results have been quite surprising and confusing. My prediction was that the ReLU model would outperform the other two by a considerable degree. Alas, I am proven wrong, and whether that has to do with a problem in implementation, my hardware, or the chosen architecture remains somewhat obscure. What we can conclude though, is that even the most sensible of predictions can surprise us in the end.

# References and Supplementary Material

Kinsley, Harrison, and Daniel Kukieła. *Neural Networks from Scratch in Python*. Harrison Kinsley, 2020.

Dietterich, Tom. “Overfitting and Undercomputing in Machine Learning.” *ACM Computing Surveys*, vol. 27, no. 3, 1995, pp. 326–327., https://doi.org/10.1145/212094.212114.

Nielsen, Michael A. *Neural Networks and Deep Learning*. Determination Press, 2015.

“L26/1 Momentum, Adagrad, RMSProp, Adam.” *Youtube*, uploaded by Alex Smola, 8 May 2019, <https://www.youtube.com/watch?v=gmwxUy7NYpA>

Hymel, Shawn. “Graph Sensor Data with Python and Matplotlib.” *Graph Sensor Data with Python and Matplotlib*, SparkFun Electronics, 23 July 2018, [https://learn.sparkfun.com/tutorials/graph-sensor-data-with-python-and-matplotlib/update-a-graph-in-real-time#](https://learn.sparkfun.com/tutorials/graph-sensor-data-with-python-and-matplotlib/update-a-graph-in-real-time%23).

Woods, Thomas. “The Logistic Function, a common Sigmoid Function.” *DeepAI*, 2020, <https://deepai.org/machine-learning-glossary-and-terms/sigmoid-function>. Accessed 8 Apr. 2022.

Stowell, Dan. “Rectifier and softplus functions.” *Wikimedia Commons*, 2017, <https://commons.wikimedia.org/w/index.php?curid=63474513>. Accessed on 8 Apr. 2022.

Adapted from Tekerek, Adem. “Proposed artificial neural network model (3-10-1).” ResearchGate, 2019, 10.14311/NNW.2019.29.013. Accessed on 8 Apr. 2022.

Katanforoosh and Gunin. “Initializing Neural Networks.” *DeepLearning.AI*, 2019, <https://www.deeplearning.ai/ai-notes/initialization/>.

Li, Haidong, et al. “Research on Overfitting of Deep Learning.” *2019 15th International Conference on Computational Intelligence and Security (CIS)*, 2020, https://doi.org/10.1109/cis.2019.00025.