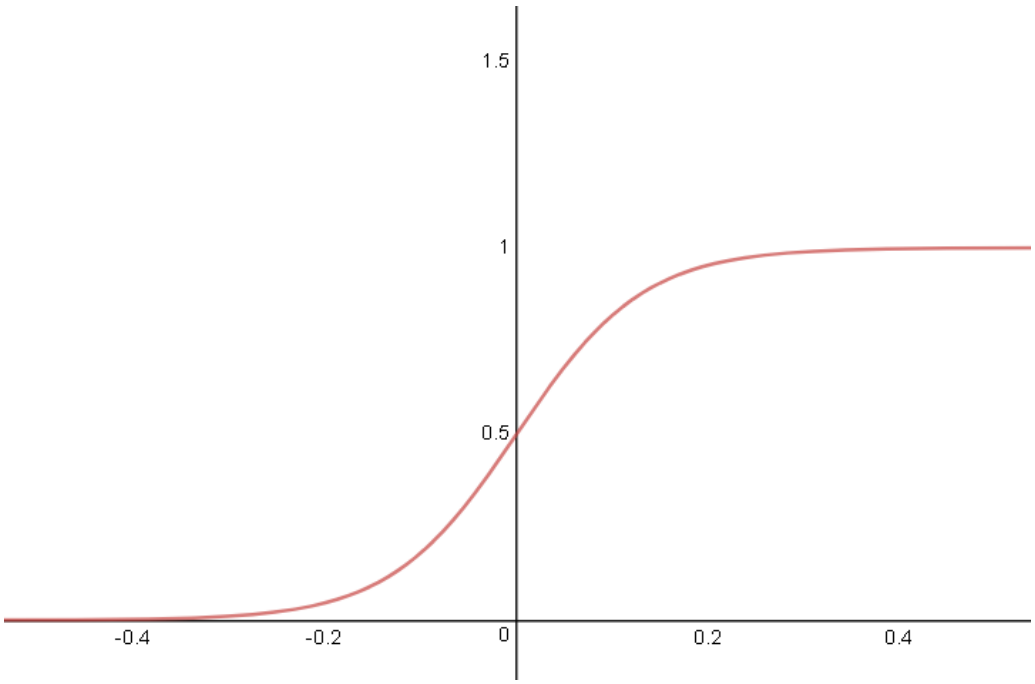


Teaching a Machine

Contents

Analysis 4
Introduction: 4
Description of Current Solution: 6
Identification of End Users: 7
Proposed Solution: 7
User Needs: 7
Acceptable Limitations: 8
Programming language: 8
Improving Runtime 9
Data Sources and Destinations 11
Data Flow Diagram 14
Objectives: 14
Documented Design 15
Overall System Design: 15
Stage 1: 15
Stage 2: 16
Stage 3: 16
Stage 4: 16
Stage 5: 17
The Maths of Simple Linear Neural Networks 18
Pseudo Code for Linear network: 23
Program Flow Chart: 24
The Problem with Linear Neural Networks 24
The Theory Behind Backpropagation 26
Description of Activation Functions: 28
The Sigmoid Function: 28
The Hyperbolic tan (tanh) function: 30
Further Theory: 31
Normalizing data 36
Weight Generation 37
Class Diagram for Node/Neuron class: 37
Functions for Backpropagation: 38
Pseudo Code for Backpropagation: 38
Design of Digit Drawing Tool: 43

Technical Solutions and Explanations: 44
Explanation of Linear Neural Network Technical Solution 44
Solution for Linear Neural Network: 52
Drawing Tool Technical Solution Explanation: 54
Drawing Tool Technical Solution: 59
Backpropagation Technical Solution Explanation 62
Backpropagation Technical Solution 76
Visualizer Technical Solution: 88
Testing: 92
Testing Screenshots: 95
Evaluation: 113
Objective Analysis 113
Possible Extensions: 114

Analysis

Introduction:
Machine Learning is a blossoming subject in computer science and is being adopted widely in industry today; it concerns the study of algorithms which, when supplied with training data, get successively better at a given task. There are many such algorithms and they vary in that data they receive and in which tasks they excel.

You can separate these algorithms into two camps:

1. **Supervised learning**. The training data contains input and the desired output. For example, an algorithm designed to separate pictures of dogs and cats would have to be supplied a large swathe of training data, as well as the species of animal in each picture. Thus whenever the machine makes an incorrect decision the components of the algorithm are changed.
2. **Unsupervised learning**. Training data is supplied without output data, only input. This is mainly used to find trends in data or to simplify data. For example, in a multivariate analysis of house prices, these kinds of algorithms could be used to determine whether some of the variables had no impact.

The algorithms themselves range from the very simple to the very complex, one of the simplest examples being linear regression (shown here).

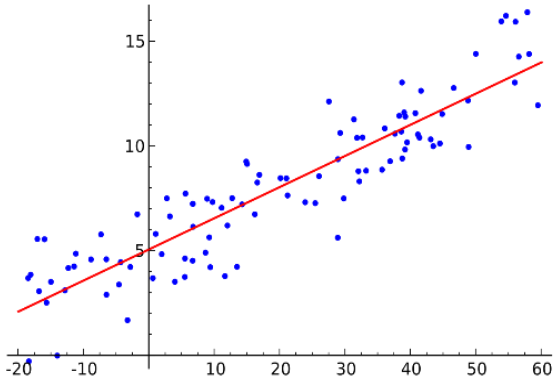


Figure 1 - An example of linear regression

Linear regression uses a technique called gradient descent to determine the average error of a given line, the better the fit the lower the error. Using squared error, we can plot a quadratic error function whose minimum is found when the gradient is zero.

The only problem is this doesn't generalise well to higher dimensional data.

A more complex and sophisticated method of learning can be found in neural networks. Neural networks are divided into layers – the first being the input layer, the last being the output layer and any layers between the two being a hidden layer. Individual layers are made up of nodes; every node in one layer being connected to every node in the next. The connections between nodes carry a weight; this weight decides the strength of the signal transferred. When training a neural network, first the training data is introduced, then the outputs are read and error calculated and finally weights are changed such to reduce error – this loop continues until favourable outputs are found. Neural networks are roughly based on animal brain anatomy, in which neurons are inhibited or excited depending on certain factors within the brain.

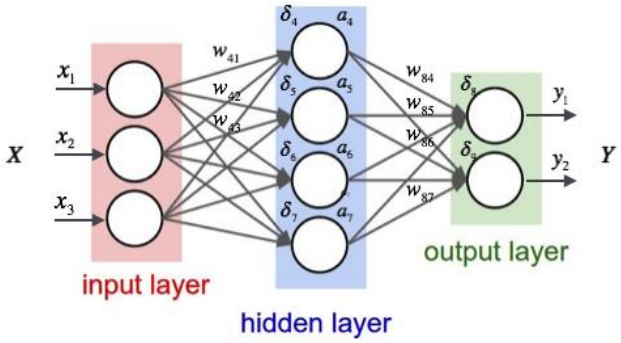


Figure 2 - An example neural network with one hidden layer

Neural networks are not suited for every task and are not easy to implement, which would explain why they are not more prominent than they are. However machine learning frameworks like Tensor flow and AmazonML API are making it easier for individuals to write sophisticated and effective machine learning solutions. The ease comes in the supply of efficient subroutines, which removes the need for a deep understanding of the algorithms themselves and instead the programmer need only understand the algorithms' capabilities. This abstraction allows for easier development of solutions. However this lower cognitive load can result in developers not being able to properly debug their code, resulting in an ineffective model.

I propose to implement a neural network for research purposes, educating myself and others about different solutions to learning problems and looking into an upcoming technology. The system would be written in python; the user would be able to decide how the method of prediction would be demonstrated and supply training data if necessary.

Description of Current Solution:

Machine learning is being used in many areas today, one of which being medicine. In 2018 a research paper was published by Google concerning an Augmented Reality Microscope capable of discerning cancer in real time.[1] The microscope takes high resolution pictures of cells placed on the stage, and feed them to a "standard off-the-shelf PC"[2] which processes the information. The image is then shown on a screen, with rings superimposed over any cancers the system may have found.

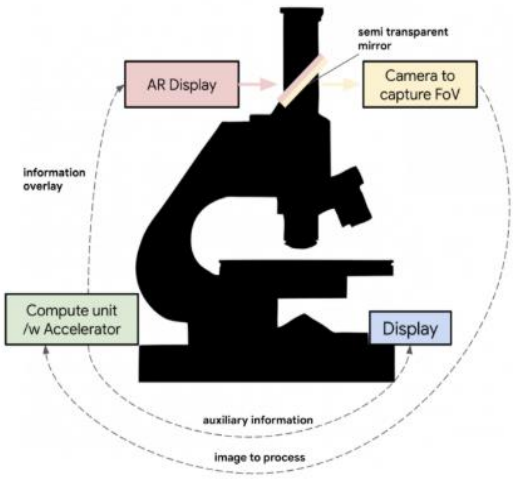


Figure 3 - Apparatus developed by Google: Image found: https://ai.googleblog.com/2018/04/an-augmented-reality-microscope.html

Training the data required pathologists to pore over thousands of images of cells to decide whether the cells were cancerous or not. This is an example of supervised learning, and can fail due to certain flaws. For example, with smaller sample sizes, the algorithm may learn the data rather than the trends. I.e. the model knows the training data well, but cannot generalise at all.

The formation of high order polynomials can also occur making the model useless for categorising anything outside of the training data. The Google team would have had to combat that problem given their (relatively for machine learning) small data set of 399 slides; a larger pattern set would make the model more accurate. Problems of smaller training sets often appear in medical machine learning, due to concerns surrounding information security – this is a difficult issue to combat, as making the images anonymous could discount some of the use of the image.[3]

The model showed a highest success rate of 0.93, which is impressive considering the relatively small size of their dataset. I would hope to be working with a much larger dataset.

[1] https://ai.googleblog.com/2018/04/an-augmented-reality-microscope.html
[2] https://arxiv.org/ftp/arxiv/papers/1812/1812.00825.pdf
[3] https://ai.googleblog.com/2018/04/an-augmented-reality-microscope.html

Identification of End Users:

Machine learning is a versatile field and can be applied in many unforeseen places; however its real boon comes in pattern recognition.

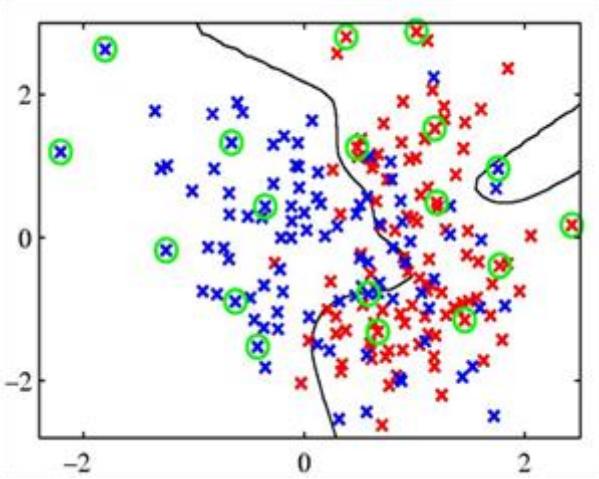


Figure 4 - An example of a support vector machine

The initial users of machine learning were (and are) programmers, people who have the knowledge and capabilities to write software capable of “learning”. The technology is now commonly available on the internet, from Amazon’s “predictive shipping” to Ai in Chess or Go. The end users of machine learning techniques are many and varied.

Proposed Solution:

My end goal is to produce a machine learning solution, which I will then use to teach a model to analyse an image of a handwritten digit and estimate what digit it has been shown.

Neural networks specifically are a good choice for several reasons. Not only are they especially versatile and capable of handling a wide range of problems, they are easy to visualise and the rudimentary workings of them (simply calculating the net value of a neural network etc.) is simple to explain.

I would also like to implement a linear neural network, to show how learning differs between the two.

User Needs:

- The user must be able to draw their own digit on the fly;
- The network will need to be visualised;
- The user should be able to test their own datasets, and try and get the system to learn from them;
- It should be easy to use, such that anyone could use it without my presence.

Acceptable Limitations:

- The loss may be lower than if I were to use third party software. However the designing of my own solution will give me a better understanding of the techniques for teaching, will allow for software designed for this task specifically and no bloated packages need be installed.
- Training neural nets takes time, should I be demonstrating the training process specifically the error will be higher than if more iterations were used. To solve this I will implement a method of loading previously trained models for complex datasets.
- In the words of a machine learning researcher: “even if a suitable number of elements and layers are chosen, in other words a solution should exist, sometimes the system will not converge”[4]. (Phil Picton) I.e. sometimes a neural network will simply not find a solution.

Programming language:

The programming language I intend to use for this project is vitally important for the success or failure of my implementation. Different languages have different strengths and weaknesses and likewise my grasp of each language differs in depth. So I will pick from a group of languages I at least have some experience of, rather than attempting to learn a language entirely from scratch.

The choices:

- Python
- C++
- Rust

Python is a dynamically typed, interpreted language that is becoming an industry standard for machine learning. What Python lacks in terms of runtime is made up for in the ease of use as well as the simplicity of understanding Python code. This is the language with which I have the most familiarity, thus no time will be needed getting to grips with its syntax.

[4] Introduction to Neural Networks p.45

C++ is a statically typed, compiled language quite apart from Python. Its blazingly fast runtime is made possible by its long compile times, and sometimes confusing syntax. I have very little experience with C++, however in the past I have used it for graphical experiments with Unity.

Rust is a statically typed, compiled language that was ranked the most loved language by developers in a survey by Stack Overflow of 100,000 developers. Its concept of ownership makes the code safer from crashes, more memory efficient but perhaps more complex to understand. I have little experience with Rust; however a chance to learn it better could benefit me later in life.

Improving Runtime

Initially I had been dissuaded from using Python due to problems of run time - machine learning requires a large amount of data and thus a large amount of data manipulation. However I discovered two very useful tools which could lessen that particular issue, namely JIT and PYPY.

JIT (Just in Time), this is a module that can be installed in Python that compiles and optimises your code. For machine learning I will be running gradient descent methods hundreds of thousands of times, compiling those methods will mean they will not need to be reinterpreted and overall execution time will be lower. However the compiler will take longer to load initially.

Another advantage of JIT is its ease of use, one simply needs to import Numba (its parent module) and simply add @jit as a header

```
from numba import jit

@jit
def factorial(number):
    if number <= 0:
        return 1
    else:
        return number * factorial(number -1)
```

PyPy is a piece of software designed to speed up Python code. It also uses a Just in Time compiler, however PyPy is not called from the code but rather the code is supplied to PyPy. PyPy is highly compatible with Python code and thus no edits need to be made to existing code.

To test these two programs against one another I ran the above recursive factorial function 100,000 times with an input of 49 firstly with the @jit header, then without the @jit header but executed by PyPy and finally with Native Python. I also used the time module to see how long each implementation took.

Implementation:

```
from numba import jit
import time


def factorial(number):
    if number <= 0:
        return 1
    else:
        return number * factorial(number -1)

number_of_runs = 100000
start_time = time.time()

for i in range(number_of_runs):
    factorial(49)

total_time = time.time() - start_time
print("Completed with a total time of", total_time)
print("Completed with an average time of", (total_time/number_of_runs))

input()
```

The only change from native to jit is adding the @jit header before the factorial function, as shown previously.

Times:

Native Python:

```
Completed with a total time of 1.7651720046997070
Completed with an average time of 1.7651720046997070e-05
```

JIT:

```
Completed with a total time of 0.26317381858825684
Completed with an average time of 2.6317381858825680e-06
```

PyPy:

```
C:\Users\magee\Desktop\Optimization\JitVsPYPY>pypy3.exe PYPY.py
Completed with a total time of 0.3593900203704834
Completed with an average time of 3.593900203704834e-06
```

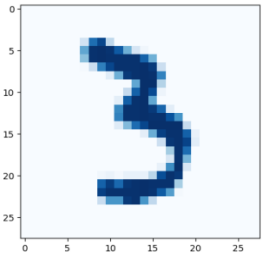
From this we observe initially that both methods complete the task far faster than Native Python, to the tune of an order of magnitude. The next thing to notice is JIT wins out with 0.263 seconds total time vs PyPy's 0.359 seconds. This difference in time is negligible here, but when larger data is used the discount could be appreciable.

In conclusion I plan to use one of these methods to speed up my eventual solution; however the code should be able to run in Native Python without Numba or PyPy, albeit slower.

Data Sources and Destinations

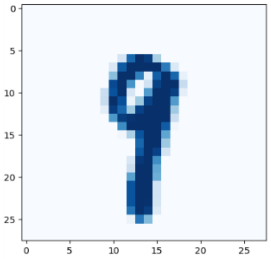
While the neural network I plan to implement will be able to learn off any data supplied to it, I think a great way of demonstrating the power of machine learning is with the MNIST dataset.

Digit 3 rendered



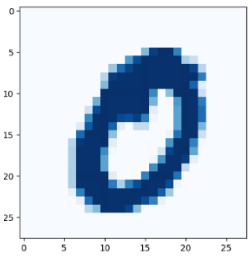
The MNIST dataset is a collection of a total of 70,000 images of handwritten digits, from 0 – 9. Of those images, 60,000 are for training the model and 10,000 are for testing the model's accuracy. That way we can see if the model is "overfitting" or not. Overfitting is when the model is exactly correct for the training data, but does not generalise well to unseen data.

Digit 9 rendered



"Overfitting happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data."[5]

Digit 0 rendered



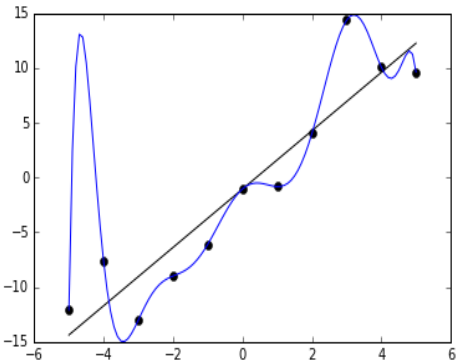


Figure 5 - An example of overfitting. In this case a straight line model would be more appropriate

Overfitting usually occurs when the training dataset is too small or too many "epochs" (iterations of the training data backwards and forwards) are completed. I will need to combat overfitting as 60,000 patterns is relatively small for machine learning.
To combat this I can limit the number of epochs or vary our weights.

[5] https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/

MNIST images are 28 by 28 pixels in size; this ensures that the processing time is minimal. Were these images HD (1080 by 1920) processing would take three orders of magnitude longer, as processing time will be one of the biggest hurdles for this project, smaller images are perfect.

The images are greyscale and centred, “the images were centred in a 28x28 image by computing the centre of mass of the pixels, and translating the image so as to position this point at the centre of the 28x28 field”[6] . According to the MNIST website the error rate improves when images are centred.

The individual pixels vary from 0 to 255 in intensity, for use in machine learning I will have to normalize this data, see page 36 for an explanation of the normalization process.

This dataset has been used by a number of neural networks before me, the lowest error rate being 0.35%. Using this data, I can find the model that works best for my implementation.

Neural Nets			
2-layer NN, 300 hidden units, mean square error	none	4.7	LeCun et al. 1998
2-layer NN, 300 HU, MSE, [distortions]	none	3.6	LeCun et al. 1998
2-layer NN, 300 HU	deskewing	1.6	LeCun et al. 1998
2-layer NN, 1000 hidden units	none	4.5	LeCun et al. 1998
2-layer NN, 1000 HU, [distortions]	none	3.8	LeCun et al. 1998
3-layer NN, 300+100 hidden units	none	3.05	LeCun et al. 1998
3-layer NN, 300+100 HU [distortions]	none	2.5	LeCun et al. 1998
3-layer NN, 500+150 hidden units	none	2.95	LeCun et al. 1998
3-layer NN, 500+150 HU [distortions]	none	2.45	LeCun et al. 1998
3-layer NN, 500+300 HU, softmax, cross entropy, weight decay	none	1.53	Hinton, unpublished, 2005
2-layer NN, 800 HU, Cross-Entropy Loss	none	1.6	Simard et al., ICDAR 2003
2-layer NN, 800 HU, cross-entropy [affine distortions]	none	1.1	Simard et al., ICDAR 2003
2-layer NN, 800 HU, MSE [elastic distortions]	none	0.9	Simard et al., ICDAR 2003
2-layer NN, 800 HU, cross-entropy [elastic distortions]	none	0.7	Simard et al., ICDAR 2003
NN, 784-500-500-2000-30 + nearest neighbor, RBM + NCA training [no distortions]	none	1.0	Salakhutdinov and Hinton, AI-Stats 2007
6-layer NN 784-2500-2000-1500-1000-500-10 (on GPU) [elastic distortions]	none	0.35	Ciresan et al. Neural Computation 10, 2010 and arXiv 1003.0358, 2010
committee of 25 NN 784-800-10 [elastic distortions]	width normalization, deslanting	0.39	Meier et al. ICDAR 2011
deep convex net, unsup pre-training [no distortions]	none	0.83	Deng et al. Interspeech 2010

Figure 6 - Data gathered from the MNIST website

Data coming out of the program will be that of saved models, i.e. a list of the data used to train the model as well as the weights, biases and overall structure of all the nodes. Users should also be able to supply their own data to be read, and then processed.

[6] http://yann.lecun.com/exdb/mnist/index.html

Python Libraries

I have decided to minimize the python libraries I will use. Certain notable exclusions are:

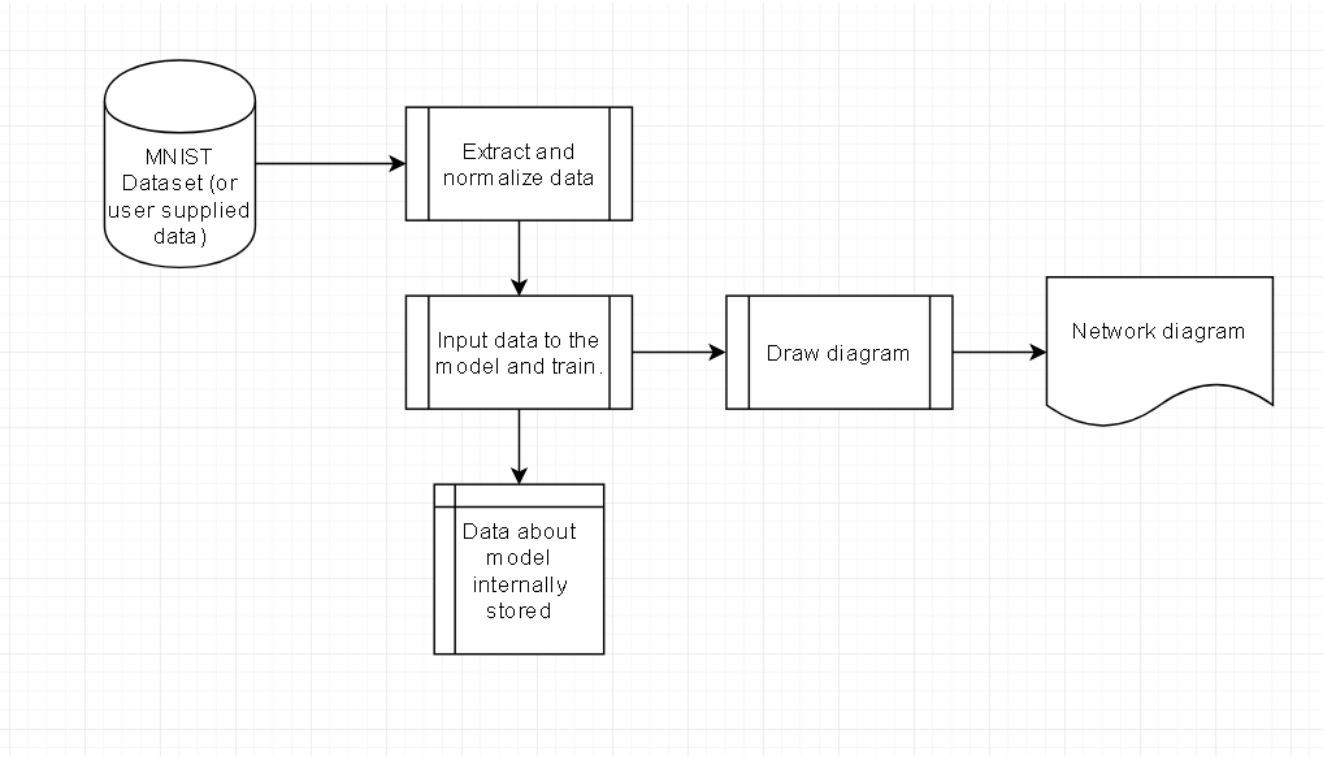
- Numpy – Usually an essential library for machine learning due to its powerful multi-dimensional array object. Comes built in with many useful (and time-efficient) functions useful for machine learning and linear algebra in general, such as dot-product, cross-product, determinants, arrangements etc.
- Sci-py – Also contains many functions for linear algebra as well as others for calculus (Calculating integrals and differentials), visualization and image processing.
- TensorFlow – An extremely powerful machine learning framework, making the procedure of building and training machine learning models extremely easy. On top if it's ease of use, it runs extremely fast – making use of multi-threading - along with the ability to run on the GPU.

My decision to avoid using these libraries is not (entirely) a modern day form of self-flagellation or self-harm, rather I hope to establish a better understanding of the theory and ideas "close to the metal" of machine learning. Through this I will be better qualified to explain machine learning techniques to others as well as go on to experiment further with machine learning in the future.

In the space of these cutting edge libraries and state-of-the-art techniques, I will be using the knowledge gained from "Introduction To Neural Networks" by Phil Picton published in 1994. A recommended book to those new to machine learning in general (not just neural networks, the book discusses many such techniques).

I will be using the notation described in this book (as explained in the Design phase) as well as some of the techniques given to improve convergence. Whenever I do either of those things, I will of course relay that to you – the reader.

Data Flow Diagram



Objectives:

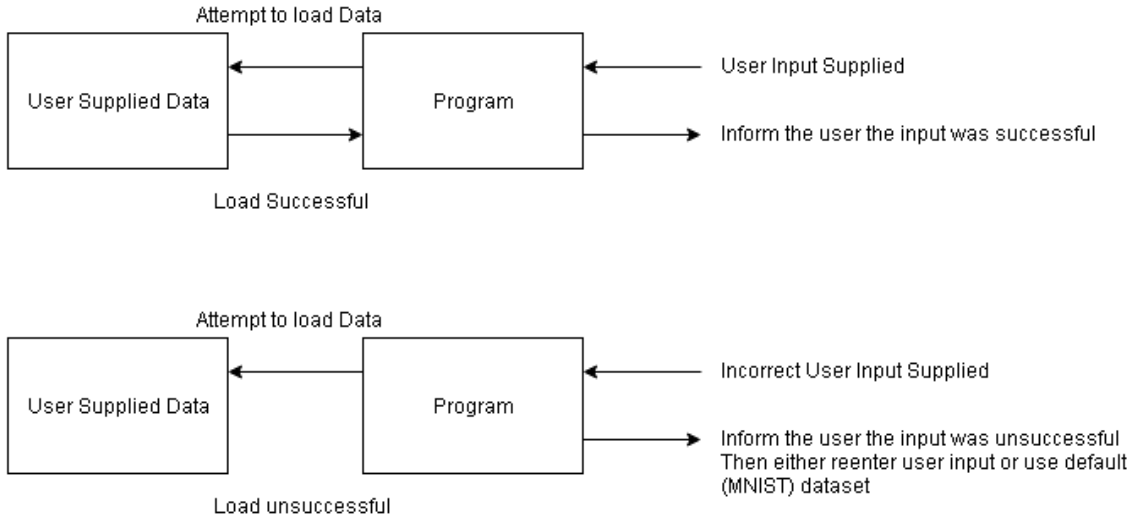
- Implement a system to generate and train neural networks, with the learning method being backpropagation
- Implement a linear neural network without backpropagation to show the difference between the two methods
- Allow the user to draw their own digits which can then be tested against a previously trained model and an output found
- Allow the user to choose data from which to learn
- Implement a method of visualizing the network. The user should be able to hide and show certain parts of the image, i.e. hide or show nodes and connections between nodes.

Documented Design

Overall System Design:

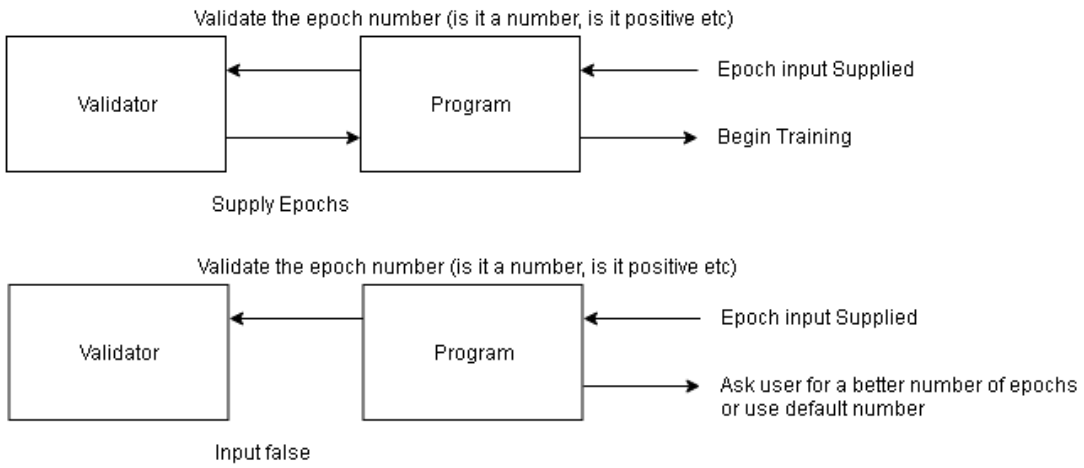
Stage 1:

Initially the user should be prompted about whether or not they want to input data. Should they decide to, the data should be found and used as input to train a model. If the data is non-existent, in the wrong format or otherwise flawed the user should be prompted to either supply a different dataset or use the default dataset.



Stage 2:

The second stage is to ask the user how many iterations they would like to run. This will decide how long the program takes to run, as well as how accurate the model is. Of course the number will need to be validated and if enter is pressed the default value should be used.

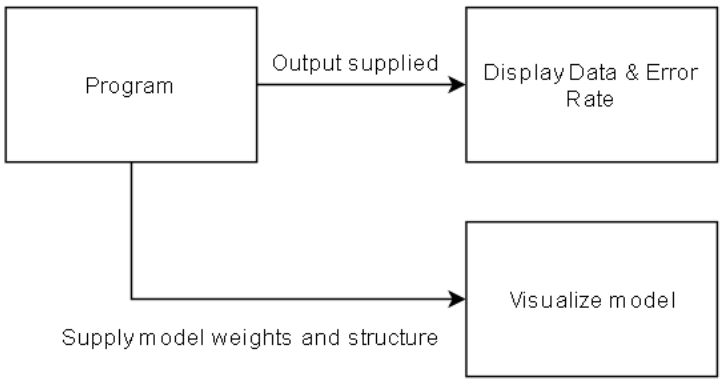


Stage 3:

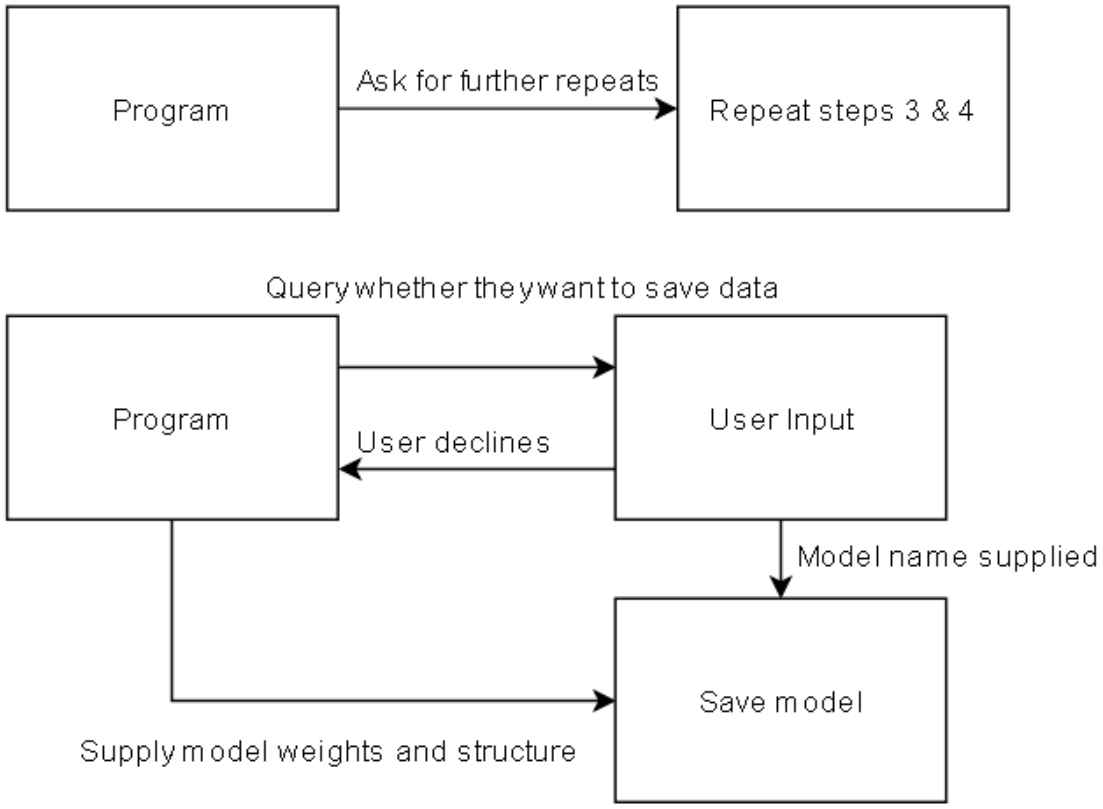
The model is generated using a neural networks, the theory of this process is explained later.

Stage 4:

Output from the model is shown, an error value is calculated and the model is visualised.



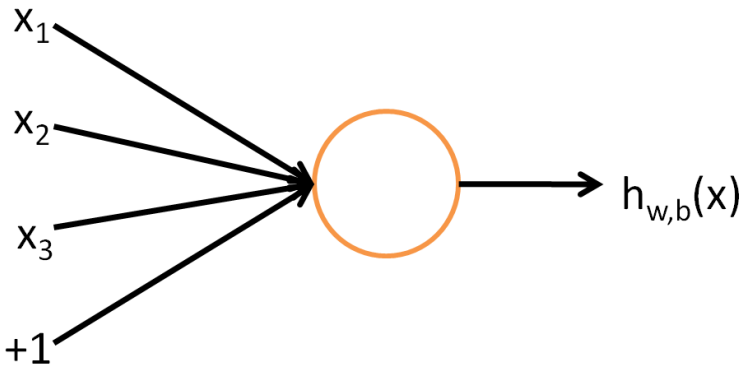
Stage 5:
The user should then be prompted whether or not they wish to continue with further iterations or to save the model and quit. Should they decide to continue for further iterations, steps 3 & 4 should be repeated. Otherwise the user should be queried about saving the model, and whether or not the user decides to save it, a temporary save should be made.



The Maths of Simple Linear Neural Networks

Neural networks are a system designed to decrease the error of a system by adjusting weights. As aforementioned connections between nodes are given a weight, the net will be the summation of the inputs to a neuron multiplied by the weights associated with each connection.

The node shown here has inputs x_1, x_2, x_3 and 1 which is the bias (discussed further on page 33). These are then multiplied with the nodes' weights to calculate the net value of a node:



$$net_p = w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 + 1 \cdot b$$

Where p is the pattern being presented to the neural network, for example in an image classifier a pattern would be one particular image and the pattern set would be the collection of all images.

In general the net would be:

$$net_p = \sum_{i=0}^{n} \left(w_i x_i \right)_p$$

The process for simple neural networks is as follows:

1. Input Data into the first layer
2. Calculate the net of all nodes (starting from the first layer and moving forward)
3. Apply the delta learning rule
4. Adjust weights
5. Repeat until convergence

A system called gradient descent will be used, by which the minimum possible loss is searched for via adjustment of weights. My error function will glean a positive quadratic, which has one and only one minimum value, so once a system is found to find that minima I will be able to train a network.

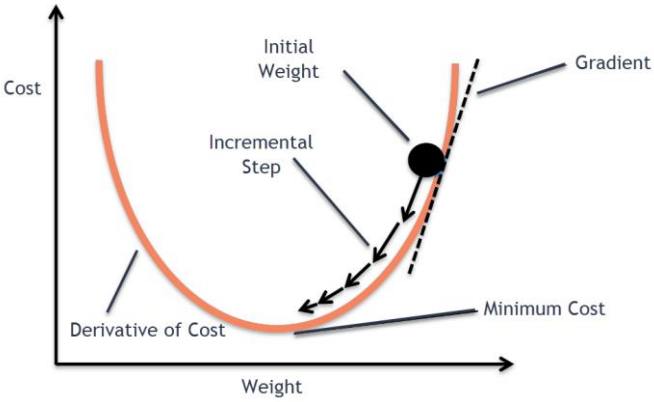


Figure 7 - Image from https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0

I can start by making the logical assumption that the adjustment of the weights should be proportional to the inputs of a node, multiplied by some function of the net value.

$$\Delta w_i \propto x_i \cdot f(net)$$

This function is the desired value (i.e. the correct output value for a piece of data), minus the net value of the neural network. This function is called the “delta” rule; I will use the delta symbol to represent it from now on.

$$f(net, d) = \delta = d_p - net_p$$

I also know that the mean error will be the sum of the error for each pattern, divided by the total number of patterns

$$E = \frac{1}{P}\sum_{p=1}^{P} e_p$$

My function for calculating error will be the square of the delta function. Squaring the function provides the benefit of only getting positive values for error, where negative error would be values below the decision surface. The negative error would have been summed along with the other errors, and decrease total error incorrectly. Thus the mean error is the sum of the squared difference between the net value of our network and the desired value. This can also be called my “cost” function.

$$e_p = \delta_p{}^2 = \left(d_p - net_p\right)^2$$

$$E = \frac{1}{P}\sum_{p=1}^{P} e_p = \frac{1}{P}\sum_{p=1}^{P}\left(d_p - net_p\right)^2$$

As I already know the net is simply the sum of the products of the weights and inputs to a node:

$$net_p = \sum_{i=0}^{n}\left(w_i x_i\right)_p$$

Therefore:

$$E = \frac{1}{P}\sum_{p=1}^{P}\left(d_p - \sum_{i=0}^{n}\left(w_i x_i\right)_p\right)^2$$

So now I have a quadratic function of the mean error. As is known, positive quadratics have only one minimum, so to get the minimum error my net value needs to be as close as possible to the desired value.

To find how much I want to change the weights I must take the partial derivative of the error with respect to the weights.

$$\frac{\partial E}{\partial w_i}$$

Where ∂E indicates a small change in the mean squared error.
And ∂w_i indicates a small change in one variable of a multivariate function.
So overall the partial derivative is the effect of a small change of one variable on the overall function.

So the change in weights should be proportional to the partial derivative of the error with respect to the weights, as the weights are the variable I can change.

$$\Delta w_i \propto \frac{\partial E}{\partial w_i}$$

To go further I can say that the change in weights is the negative of some positive constant k multiplied by the partial derivative. The reason for its sign is obvious, should the error be increasing because of an increase in weights then I would want to negatively change the weights.

$$\Delta w_i = -k \cdot \frac{\partial E}{\partial w_i}$$

We can then compute the partial derivative and find:

$$\frac{\partial E}{\partial w_i} = \frac{1}{P}\sum_{p=1}^{P}\frac{\partial e_p}{\partial w_i}$$

Using a simple example of the chain rule, namely:

$$\frac{\partial e_p}{\partial w_i} = \frac{\partial e_p}{\partial \delta_p}\cdot\frac{\partial \delta_p}{\partial w_i}$$

Therefore:

$$\frac{\partial E}{\partial w_i} = \frac{1}{P}\sum_{p=1}^{P}\frac{\partial e_p}{\partial w_i} = \frac{1}{P}\sum_{p=1}^{P}\frac{\partial e_p}{\partial \delta_p}\cdot\frac{\partial \delta_p}{\partial w_i}$$

We do this as we have a function of error that is in terms of delta and a function delta that is in terms of weights. Thus the derivative can easily be computed:

$$e_p = {\delta_p}^2 = \left(d_p - net_p\right)^2$$

$$\frac{\partial e_p}{\partial \delta_p} = 2\delta_p$$

So if I substitute that value into our partial derivative for error, we get:

$$\frac{\partial E}{\partial w_i} = \frac{1}{P}\sum_{p=1}^{P}2\delta_p\cdot\frac{\partial \delta_p}{\partial w_i}$$

Then I can use the chain rule again to garner:

$$\frac{\partial \delta_p}{\partial w_i} = \frac{\partial \delta_p}{\partial net_p}\cdot\frac{\partial net_p}{\partial w_i}$$

And as:

$$f(net, d) = \delta = d_p - net_p$$

We can compute the derivative and find:

$$\frac{\partial \delta_p}{\partial net_p} = -1$$

And finally, as:

$$net_p = \sum_{i=0}^{n} \left(w_i x_i \right)_p$$

Then:

$$\frac{\partial net_p}{\partial w_i} = x_{ip}$$

$$\frac{\partial \delta_p}{\partial w_i} = \frac{\partial \delta_p}{\partial net_p} \cdot \frac{\partial net_p}{\partial w_i} = -x_{ip}$$

So at last we can combine our chain rules and find:

$$\frac{\partial E}{\partial w_i} = -\frac{1}{P} \sum_{p=1}^{P} 2\delta_p \cdot x_{ip} = -\frac{2}{P} \sum_{p=1}^{P} \delta_p \cdot x_{ip}$$

$$\Delta w_i = \frac{2k}{P} \sum_{p=1}^{P} \delta_p \cdot x_{ip}$$

Thus by adjusting the weights proportionally to the mean value of $\delta_p x_{ip}$ I will garner improvements in the accuracy of the system.
I rename the constant 2k and gain a simplified form:

The value of η decides the size of the “jumps” in gradient descent, i.e. how far down the quadratic you move when weights are changed. Thus η should be large enough to ensure convergence doesn’t take too long, yet small enough that the weights settle down and convergence is actually reached.

$$\Delta w_i = \frac{\eta}{P} \sum_{p=1}^{P} \delta_p \cdot x_{ip}$$

In order to implement a network like this in Python, certain basic functions will need to be implemented:

- A function to calculate the net value of the network
- A function to calculate the delta rule for a certain pattern
- A function to calculate our change in weights
- A function to change the weights of our nodes

Pseudo Code for Linear network:

Net value Pseudo Code:

```
FUNC get_net(data_value, node)
        net = 0
        FOR weight IN node
                net = net + data_value * node.weight[data_value]
```

Delta rule Pseudo Code:

```
FUNC get_delta_rule(desired_data, net_value)
        RETURN desired_data – net_value
```

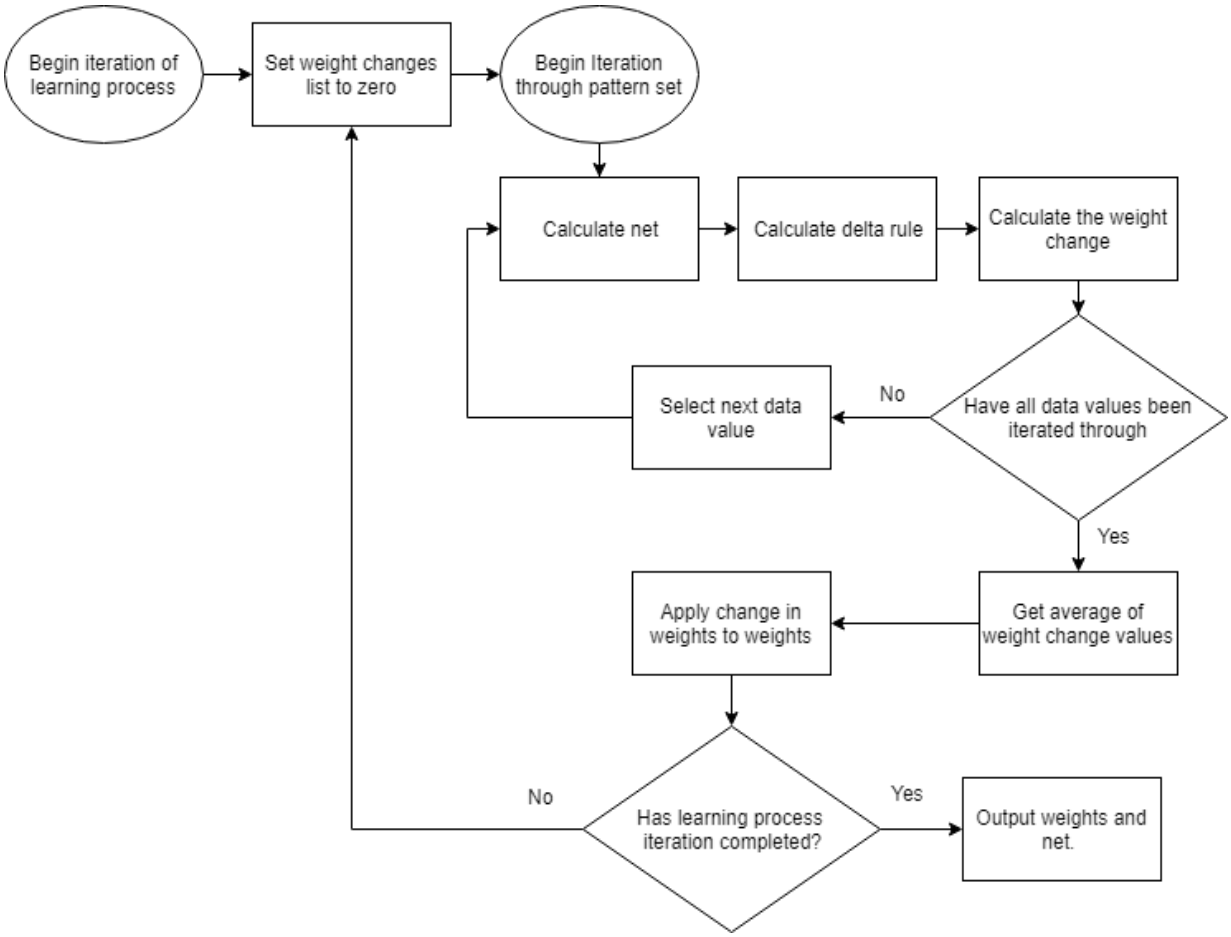
Calculate Change in Weights Pseudo Code:

```
FUNC get_change_in_weights(data_list, delta_rule, offset_value)
        weight_changes = []
        FOR data IN data_list
                weight_changes = delta_rule * offset_value * data
        RETURN weight_changes
```

Change Weights Pseudo Code

```
FUNC change_weights(initial_weights, weight_changes)
        FOR weight IN initial_weights
                Initial_weights[POSITION(weight)] +=
weight_changes[POSITION(weight)]
        RETURN initial_weight
```

Program Flow Chart:



The Problem with Linear Neural Networks

This solution only concerns a single layer, however, resulting in the terrible consequence that it can only classify linearly separable input spaces. A linearly separable input space is one which can be separated into its classes by a single line, shown here, crosses and circles can be separated using the line L. This line is known as the “decision surface”.
Our network is defining a decision surface to correctly classify data.

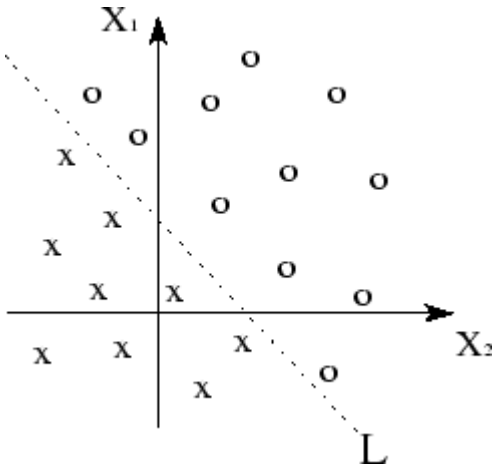


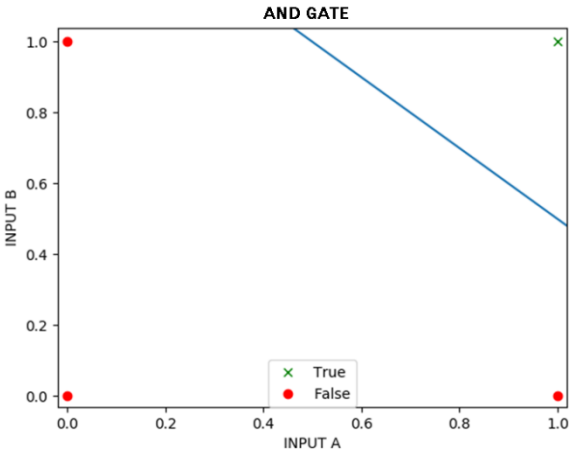
Figure 8 - Example of a decision surface

If we are to look at the input space for an AND gate, we can immediately see it’s linearly separable by the line:

$$y = 3/2 - x$$

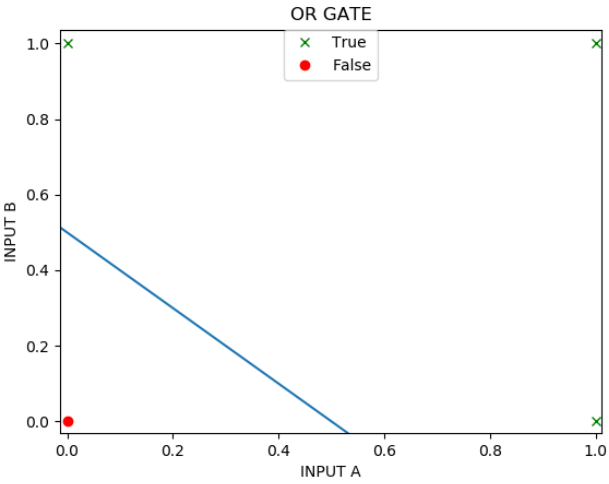
Where the points are:

$$(0,0)\ \ (0,1)\ \ (1,0)\ \ (1,1)$$



Likewise an OR gate can be separated, this time by the line:

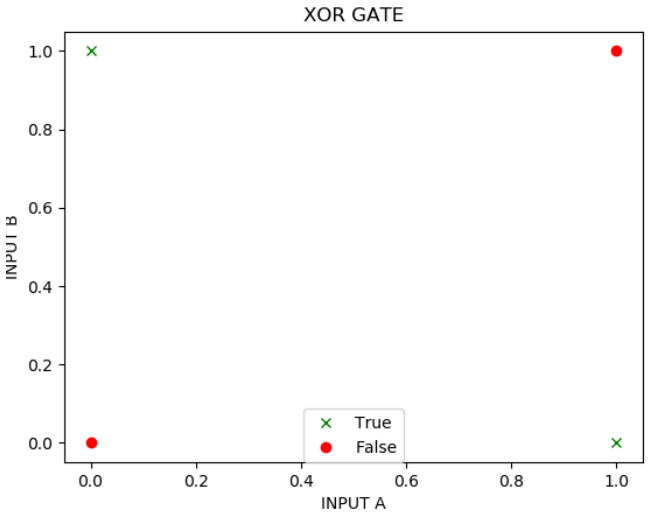
$$y = 1/2 - x$$



Where the points are the same, but clearly the classification is that of an AND gate. Either of these decision surfaces could change, for our AND gate, the surface could have a y intercept of between 1 and 2. While our OR gate's y intercept could also vary:

$$1 \leq c_{AND} < 2$$
$$0 \leq c_{OR} < 1$$

Where c is the y intercept. If you recall, we are representing our lines in the form $y = mx + c$, where m is the gradient of our line (in this case -1) and c is the y intercept.



However we run into a problem when we try to learn an XOR or exclusive OR gate. When we display those inputs and outputs on our graph we find:

There is no possible decision surface (in the form of a straight line) that would correctly classify these inputs. To classify non-linear functions, we need a more sophisticated method of learning, as well as a non-linear activation function.

The Theory Behind Backpropagation

Activation Functions:
Activation functions in neural networks are functions through which the net value of a node is filtered and the output of that node calculated.

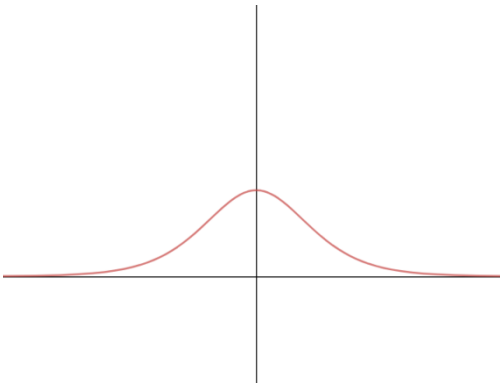


Figure 9 - A graphed derivative of the Sigmoid Function

For example, for an activation function f which takes an input of the net value, we have:

1) $$output = f\left(net_p\right)$$

Should we decide to differentiate this, we would have:

2) $$Derivative\ of\ \ output = \frac{\mathrm{d}f}{\mathrm{d}net_p}$$

We will continue to use the mean squared error function, so we remember our change of weights is:

3) $$\Delta w_i = -k \cdot \frac{\partial E}{\partial w_i}$$

Where:

4) $$E = \frac{1}{P}\sum_{p}^{P} e_p$$

But this time, our output value in the mean squared error will be that of our activation function:

5) $$e_p = \left(d_p - f\left(net_p\right)\right)^2$$

Where, as before, the values have been squared, this ensures that relatively large values for error are even larger and small values are made smaller.

Of course the net value is still being calculated as before, its partial derivative also remains the same:

6) $$net_p = \sum_{i=0}^{n} w_i x_i$$ And: $$\frac{\partial net_p}{\partial w_i} = x_{ip}$$

Now, if we make use of the Chain rule again, we find:

7) $$\frac{\partial E}{\partial w_i} = \frac{1}{P} \sum_{p=1}^{P} \frac{\partial e_p}{\partial w_i} = \frac{1}{P} \sum_{p=1}^{P} \frac{\partial e_p}{\partial f\left(net_p\right)} \cdot \frac{\partial f\left(net_p\right)}{\partial w_i}$$

And if we differentiate (partially) figure 5 with respect to our function of net, we find:

8) $$\frac{\partial e_p}{\partial f\left(net_p\right)} = -2\left(d_p - f\left(net_p\right)\right)$$

We can then substitute that into figure 7 to find:

9) $$\frac{\partial E}{\partial w_i} = \frac{1}{P} \sum_{p=1}^{P} -2\left(d_p - f\left(net_p\right)\right) \cdot \frac{\partial f\left(net_p\right)}{\partial w_i}$$

Then if we use the chain rule again:

10) $$\frac{\partial E}{\partial w_i} = -\frac{2}{P} \sum_{p=1}^{P} \left(d_p - f\left(net_p\right)\right) \cdot \frac{\partial f\left(net_p\right)}{\partial net_p} \cdot \frac{\partial net_p}{\partial w_i}$$

No
w we can substitute figure 6 into figure 10 to find:

11)

$$\frac{\partial E}{\partial w_i} = -\frac{2}{P}\sum_{p=1}^{P} x_{ip} \cdot \left(d_p - f\left(net_p\right)\right) \cdot \frac{\partial f\left(net_p\right)}{\partial net_p}$$

To complete our function of error, we need to decide on which activation function to use. Different activation functions have different strengths and weaknesses, the activation function can decide the speed and accuracy of training.

Description of Activation Functions:

The Sigmoid Function:

The Sigmoid Function is a variant of the logistic function. Logistic functions are those of the form:

$$f(x) = \frac{L}{1 + e^{-k\left(x - x_0\right)}}$$

Where:

- L is the curve's maximum value
- K defines the steepness of our curve (Can also be said to shrink or stretch the function, but the effect is the same)
- E is Euler's number
- x_0 is the position of the centre of the logistic function (on the x axis)

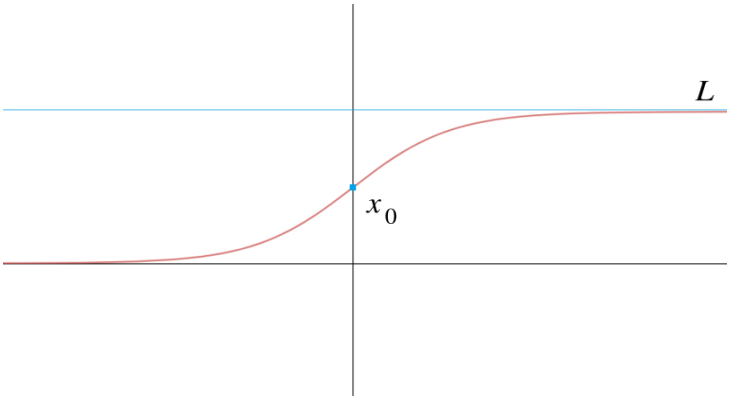


Figure 10 - The Sigmoid Function

As the input to the function varies to ∞^+, logistic functions tend towards their maximum value L and while inputs vary to ∞^-, these functions tend towards 0.

"The initial stage of growth is approximately exponential (geometric); then, as saturation begins, the growth slows to linear (arithmetic), and at maturity, growth stops."[7]

[7] https://en.wikipedia.org/wiki/Logistic_function

This varying rate of increase is perfect for machine learning, when the net value is relatively small you want it to increase exponentially to reach the minima of a function however you don't want the net to grow so much that the minima is "jumped" over, thus the growth dies down to linear and then zero.

The Sigmoid Function is a form of logistic function whose maximum value and "steepness" are unity and its centre position is zero.
It is differentiable at all points and defined as:

12)
$$S(x) = \frac{1}{1 + e^{-x}}$$

Where its derivative[8] is:

13)
$$S'(x) = \frac{e^{-x}}{\left(1 + e^{-x}\right)^2}$$

(Note that s'(x) simply means the derivative, simply a different method of notation) The final thing to note is that figure 13 is equivalent to:

14)
$$S'(x) = S(x)\,(1 - S(x))$$

This will speed up calculation as we need only calculate S'(x) and then use that to find S'(x).

[8] http://kawahara.ca/how-to-compute-the-derivative-of-a-sigmoid-function-fully-worked-example/

The Hyperbolic tan (tanh) function:

Tanh is a hyperbolic curve, which are curves that have similar names and properties to trigonometric functions but are defined exponentially. That is with Euler's number raised to powers.

$$sinh(x) = \frac{e^{x} - e^{-x}}{2}$$

$$cosh(x) = \frac{e^{x} + e^{-x}}{2}$$

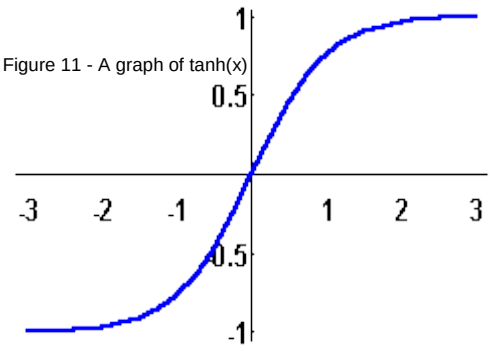


Figure 11 - A graph of tanh(x)

Dividing sinh x by cosh x gives us tanh x! This function is visually similar to the Sigmoid Function but is bounded not from (0, 1) but from (-1, 1) along the x axis. This of course gives the possibility of nodes outputting negative values, which is impossible with the Sigmoid Function.

According to "Efficient BackProp" by Yann Lecun, "Symmetric Sigmoids such as hyperbolic tangent often converge faster than the standard logistic function"[9] – note that the standard logistic function refers to the aforementioned Sigmoid Function.

The tanh function is defined as:

15) $$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

And its derivative, quite nicely, turns out to be:

16) $$\frac{\mathrm{d}}{\mathrm{d}x}(tanh(x)) = 1 - (tanh(x))^{2}$$

[9] http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

Further Theory:
If we now continue, with our activation function as the sigmoid function for demonstration, figure 11 becomes (with S as our sigmoid function):

$$\frac{\partial E}{\partial w_i} = -\frac{2}{P}\sum_{p=1}^{P}\left(d_p - S\left(net_p\right)\right)S\left(net_p\right)\left(1 - S\left(net_p\right)\right)x_{ip} \quad 17)$$

We can then substitute into figure 3 to find:

$$\Delta w_i = \frac{2k}{P}\sum_{p=1}^{P}\left(d_p - S\left(net_p\right)\right)S\left(net_p\right)\left(1 - S\left(net_p\right)\right)x_{ip} \quad 18)$$

So we may define our **delta rule** as:

$$\delta_p = \left(d_p - S\left(net_p\right)\right)S\left(net_p\right)\left(1 - S\left(net_p\right)\right) \quad 19)$$

And finally we return to a similar form as that of our linear neural network, namely:

$$\Delta w_i = \frac{\eta}{P}\sum_{p=1}^{P} x_{ip}\delta_p \quad 20)$$

This expression is usually approximated to:

$$\Delta w_i = \eta\, x_{ip}\delta_p \quad 21)$$

This works for neurons in the output layer, however neurons in the hidden layer who are not directly connected to the output need a

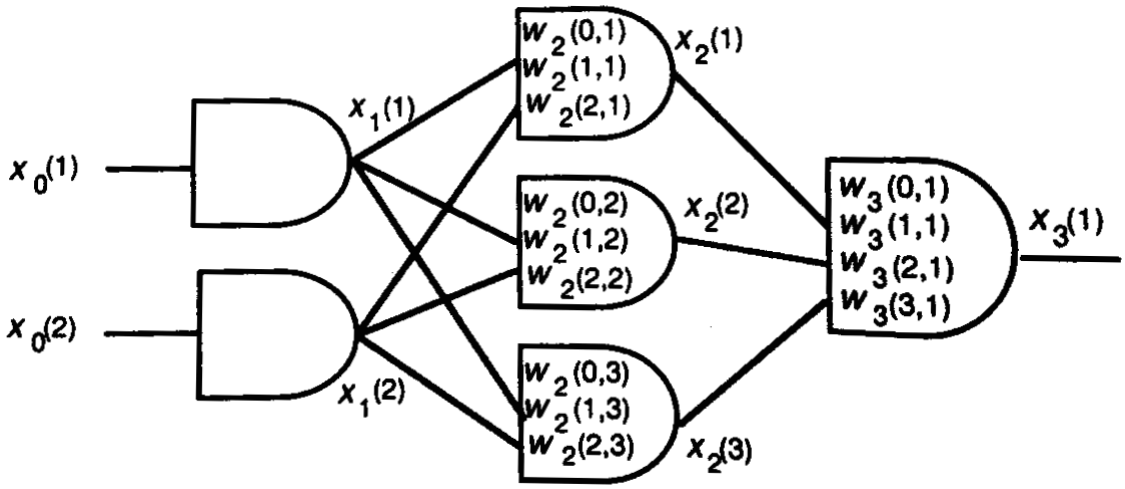


Figure 12 - Example neural network

different definition for the delta rule.Consider a 3 layer neural net with two input nodes, one output node andthree nodes in the hidden layer. [10]

For the notation here x is the inputs and outputs of each node,

$$x_0(0)\ ,\ x_0(1)\ \ldots\ x_0(n)$$

Figure 13 - An example of input data going to a node with "n" inputs

Where the subscripted value denotes which layer of nodes from which x is output, while the bracketed value refers to which row the connection resides in. I.e. $x_2(1)$ refers to an output from a node in the second layer and first row. $X_0(n)$ refers to input data.
The notation for weights is as such:

$$w_1(0,2)\ ,\ w_1(1,2)\ \ldots\ w_1(n,2)$$

The subscripted value refers to which layer the weight resides in, the first bracketed value refers to which of the neurons in the previous row is being weighted and the second bracketed value refers to which row the weight resides in. For example w_2 (2, 1) refers to a weight on the second layer, connected to the second node of the previous layer and on the first row of its layer.

[10] Image from Introduction to Neural Networks by Phil Picton

Any weight with a zero as the first bracketed value is an offset node:

$$w_1(0,1),\ w_1(0,2)\ \ldots\ w_1(0,n)$$

Figure 14 - The offset weights for a hidden layer of size 'n'

In effect, a bias value allows you to shift the activation function along the x axis, which may be critical for successful learning. Changing the weights has the effect of varying the steepness of the curve; however that alone cannot classify all our data.

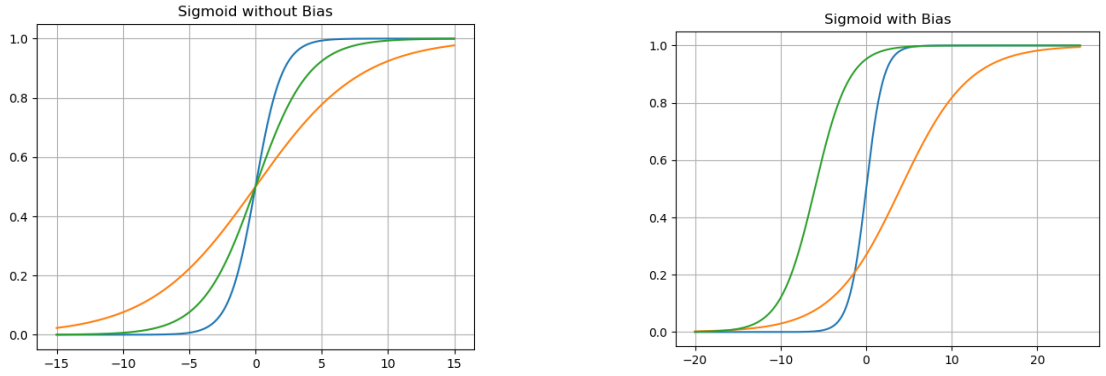


Figure 15 - Example of Sigmoid with/without biases

Consider the output of the network shown in figure 11. It will be a non-linear function of its weights multiplied by the output of the previous layer:

22)
$$x_3(1) = f\left(\sum_{i=0}^{3}\left(w_3(i,1)\cdot x_2(i)\right)\right)$$

Likewise the output of the second node in the previous layer will be a non-linear function of its weights and inputs.

23)
$$x_2(2) = f\left(\sum_{i=0}^{2}\left(w_2(i,2)\cdot x_1(i)\right)\right)$$

So if we were to take a partial derivative of the error with respect to the first weight of that node and use the chain rule we would garner:

24) $$\frac{\partial E}{\partial w_2(1,2)} = \frac{\partial E}{\partial x_3(1)} \cdot \frac{\partial x_3(1)}{\partial w_2(1,2)}$$

We do this as we cannot directly measure the error from inside the hidden layer; but we can from our final output node.
We then use the chain rule again to find:

25) $$\frac{\partial E}{\partial w_2(1,2)} = \frac{\partial E}{\partial x_3(1)} \cdot \frac{\partial x_3(1)}{\partial x_2(2)} \cdot \frac{\partial x_2(2)}{\partial w_2(1,2)}$$

Again we are trying to get our partial derivatives in terms of our final output, from which we can find the error. Which we do to find:

26) $$\frac{\partial E}{\partial x_3(1)} = -2\left[d - x_3(1)\right]$$

We can also partially differentiate $x_3(1)$ with respect to $x_2(2)$, this is simply done when considering Equation 24 & 25:

27) $$\frac{\partial x_3(1)}{\partial x_2(2)} = w_3(2,1) \cdot x_3(1) \cdot \left[1 - x_3(1)\right]$$

Likewise for $x_2(2)$:

28) $$\frac{\partial x_2(2)}{\partial w_2(1,2)} = x_1(1) \cdot x_2(2) \cdot \left[1 - x_2(2)\right]$$

Then substituting all of that into Equation 27 we get:

29) $$\frac{\partial E}{\partial w_2(1,2)} = -2x_1(1)\cdot x_2(2)\cdot\left[1 - x_2(2)\right]\cdot w_3(2,1)\cdot x_3(1)\cdot\left[1 - x_3(1)\right]\cdot\left[d - x_3(1)\right]$$

If we then sub that into Equation 3, we find:

30) $$\Delta w_2(1,2) = 2kx_1(1)\cdot x_2(2)\cdot\left[1 - x_2(2)\right]\cdot w_3(2,1)\cdot x_3(1)\cdot\left[1 - x_3(1)\right]\cdot\left[d - x_3(1)\right]$$

We can then simplify, using our previously defined delta rule (Equation 21), we find:

31) $$\Delta w_2(1,2) = 2kx_1(1)\cdot x_2(2)\cdot\left[1 - x_2(2)\right]\cdot w_3(2,1)\cdot\delta_3(1)$$

Where:

32) $$\delta_3(1) = x_3(1)\cdot\left[1 - x_3(1)\right]\cdot\left[d - x_3(1)\right]$$

We can then go further:

33) $$\Delta w_2(1,2) = 2kx_1(1)\cdot\delta_2(1)$$

Where:

34) $$\delta_2(1) = x_2(2)\cdot\left[1 - x_2(2)\right]\cdot w_3(1,2)\cdot\delta_3(1)$$

For more output neurons, the error E would be the sum:

35) $$E = \left[d_1 - x_3(1)\right]^2 + \left[d_2 - x_3(2)\right]^2 + \ldots\left[d_m - x_3(m)\right]^2$$

So the derivative of the error with respect to the output would be:

36) $$\sum\frac{\partial E}{\partial x_3(i)} = -2\sum\left[d_i - x_3(i)\right]$$

So the general form of the delta rule for element q in hidden layer p is:

37) $$\delta_p(q) = x_p(q) \cdot \left[1 - x_p(q)\right] \cdot \sum w_{p+l}(q, i)\, \delta_{p+l}(i)$$

So to apply backpropagation we must first calculate the net value of all nodes (starting from the input nodes) – to ultimately calculate the error at the output. Then begin at the output and calculate the delta rule for each node and finally find the change in weights.

Normalizing data

We will normalize our data so its mean is 0 and its standard deviation is 1. We do this because "Convergence is usually faster if the average of each input variable over the training set is close to zero"[11]

$$X_{changed} = \frac{X - \mu}{\sigma}$$

The equation for normalizing with this method will be applied to each pixel of the image, with the mean and standard deviation being of the image as a whole.

Normalization allows us to place our values in a similar scale, for example were we to do a multi variate analysis, two of our variables could be number of children and annual earnings. For this model, number of children will vary 0-10 while income will vary from 20,000 to 1,000,000 (any values outside of those scales will be outliers and removed from consideration). Were we analysing how those act as predictors for another variable, say age, annual incomes larger values would mean it will influence the results of our model more. However this may not be the case, normalization solves this problem!

Example of normalized vs un-normalized data (Age used here).

Age (pre normalization)	Income (pre normalization)	Age (post normalization)	Income (post normalization)
62	222850	0.366516325	-0.1641226432
77	459629	1.109454822	-1.799443127
51	518465	-0.1783052392	-0.8294025225
41	347002	-0.6735975702	0.2417315527
76	377468	1.059925588	-0.6182036537
66	779259	0.5646332574	1.709302437

[11] http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

64	622322	0.4655747912	0.115591351
63	494420	0.4160455581	-0.5995218833
6	164185	-2.407120729	0.5124986467
40	286819	-0.7231268033	1.431569842

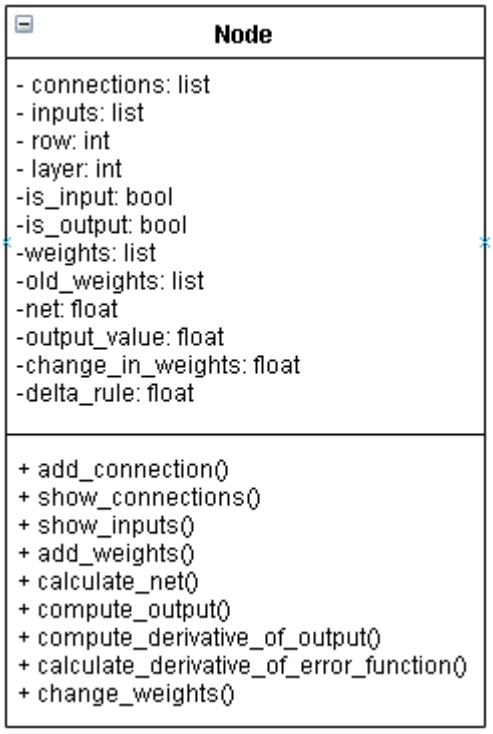
Weight Generation

$$\sigma_w = m^{-1/2}$$

We will be using the Xavier method of weight generation. The method is, "…weights should be randomly drawn from a distribution with mean zero and standard deviation $m^{-0.5}$ where m is the fan-in (the number of connections feeding into a node)"[12]

This ensures that the standard deviation of outputs from all the nodes is approximately 1. This in turn ensures that the values don't become too large and out of control.

Class Diagram for Node/Neuron class:



[12] http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

Functions for Backpropagation:

- Node Class
- add_weights – This will change the weights of individual “Node” class instantiations (i.e. individual neurons).
- calculate_net – This will calculate the net value for individual neurons.
- compute_output – This will calculate the output of individual neurons via the aforementioned activation functions.
- compute_derivative_of_output – This will calculate the derivative of the output functions.
- calculate_derivative_of_error_function – This will calculate the derivative of our error function.
- change_weights – This will update the weights of individual neurons.
- network_creator – This will create the necessary number of instations of the “Node” class, as different layers will have a varying number of nodes. It will also decide if a neuron is an input or output neuron.
- connect_all_nodes – This will connect every neuron layer.
- generate_weights – This will initialise the weights of all the neurons.
- Runner – This will organise the running of all of the functions to ensure learning.

Pseudo Code for Backpropagation:

Node Class

```
CLASS Node:

    FUNC __init__(self, row, layer, is_input=False, is_output=False):

        self.connections = []   //Stores the nodes to output to
        self.inputs = []   //Stores the node’s inputs
        self.row = row
        self.layer = layer
        self.is_input = is_input   //Stores whether or not the node is an input node
        self.is_output = is_output
        self.weights = []   //Stores the node’s weights
        self.old_weights = []   //Stores the node’s old weights for use in backprop
        self.net = 0
        self.output_value = 0   //Stores the value of the nodes net ran through it’s
```

```
activation function
        self.change_in_weight = 0
        self.delta_rule = 0
```

add_weights (A method of the Node class):

```
FUNC add_weights(self, weights)   //A simple method to change the weights of a instance of
Node
       IF NOT self.is_input:
                      self.weights = weights
       ENDIF
ENDFUNC
```

calculate_net (A method of the Node class):

```
FUNC calculate_net(self, data_input)
       self.net = 0
       IF NOT self.is_input:
                      FOR input_node_no IN LEN(self.inputs):   //Iterates through its
inputs and multiplies each input by the corresponding weight
                             self.net = self.net + self.inputs[input_node_no] *
self.weights[input_node_no + 1]
               self.net = self.net + self.weights[0]
       ELSE
               self.net = data_input   //Input nodes store their data as the net
       self.compute_output()
     ENDIF
ENDFUNC
```

compute_output (A method of the Node class):

This is dependent on which activation function we use, so I will give the pseudo code for each one.

```
FUNC compute_output(self) \\Sigmoid
IF self.is_input:
               self.output_value = self.net   //Any input nodes have an output value of
their data, which is set to their net in calculate net
          ELSE:
               self.output_value = 1 / (1 + e^(-self.net))   //Simply calculates the result
of the sigmoid function
       ENDIF
ENDFUNC


FUNC compute_output(self) \\Tanh
IF self.is_input:
               self.output_value = self.net
          ELSE:
               self.output_value = tanh(self.net)
       ENDIF
ENDFUNC

FUNC compute_output(self) \\Tanh
IF self.is_input:
               self.output_value = self.net
          ELSE:
```

```
                self.output_value = MAX(0, self.net)
        ENDIF
ENDFUNC
```

compute_derivative_of_output (Inside the Node class):

Again dependent on which activation function is used:

```
FUNC compute_derivative_of_output(self): //Sigmoid
        RETURN self.output_value * (1 - self.output_value)
ENDFUNC


FUNC compute_derivative_of_output(self): //tanh
        RETURN 1 - (self.output_value)^2
ENDFUNC


FUNC compute_derivative_of_output(self): //ReLU
        IF self.net > 0:
                RETURN 1
        ELSE:
                RETURN 0
        ENDIF
ENDFUNC
```

calculate_derivative_of_error_function (A method of the Node class):

```
FUNC calculate_derivative_of_error(self, desired_value)
        RETURN (desired_value - self.output_value)   //Simply calculates the derivative of
the error function
```

change_weights (A method of the Node class):

```
FUNC change_weights(self, desired_value, learning_coefficient):
   change_in_weights = []

   IF self.is_output
      IF self.row == desired_value
         desired_value = 0
      ELSE
         desired_value = 0
      ENDIF
      self.old_weights = COPY(self.weights)

      self.delta_rule = self.compute_derivative_of_output() *
self.calculate_derivative_of_error_function(desired_value)

      FOR node IN self.inputs
```

```
            change_in_weights.APPEND(node.output_value*self.delta_rule*learning_coefficient)
        ENDFOR

        FOR change_no IN LEN(change_in_weights)
            self.weights[change_no] = (change_in_weights[change_no] +
self.weights[change_no+1] )
        ENDFOR

        self.weights[0] = self.weights[0] + self.delta_rule * learning_coefficient
    ELSE
        self.old_weights = COPY(self.weights)
        propagation_value = 0

        FOR node IN self.connections
            propagation_value = propagation_value + [node.old_weights[self.row+1] *
node.delta_rule
        ENDFOR

        self.delta_rule = self.compute_derivative_of_output() * propagation_value

        FOR node IN self.inputs
            change_in_weights.APPEND(node.net* self.delta_rule*learning_coefficien)
        ENDFOR

        FOR change_no IN LEN(change_in_weights)
            self.weights[change_no] = (change_in_weights[change_no] +
self.weights[change_no+1] )
        ENDFOR

        self.weights[0] = self.weights[0] + self.delta_rule * learning_coefficient
```

network_creator:

```
FUNC network_creator(node_list, structure_list)
    layer = 0
    FOR number_of_nodes IN structure_list
        IF layer == 0
            row = number_of_nodes
            WHILE row > 0
                node_list.APPEND(Node(row=row, layer=layer, is_input=True))
                row = row -1
            ENDWHILE

        ELIF layer == LEN(structure_list)-1
            row = number_of_nodes
            WHILE row > 0
                node_list.APPEND(Node(row=row, layer=layer, is_input=False, is_output =True))
                row = row -1

            ENDWHILE

        ELSE
            row = number_of_nodes
            WHILE row > 0
                node_list.APPEND(Node(row=row, layer=layer))
                row = row -1
```

```
            ENDWHILE
        ENDIF
        layer = layer + 1
    ENDFOR
```

Connect_all_nodes(nodes_list):

```
FUNC connect_all_ nodes_list
    FOR node IN nodes_list
        IF NOT node.is_output
            FOR other_node IN nodes_list
                IF other_node.layer == node.layer + 1
                    node.add_connections(other_node.row, other_node.layer, nodes_list)
                ENDIF
            ENDFOR
        ENDIF
    ENDFOR
```

generate_weights:

```
FUNC generate_weights(nodes_list)
    FOR node IN nodes_list
        IF node.layer != 0
            FOR inputs IN LEN(node.inputs)+1
                node.weights.append(random.gauss(0, len(node.inputs)**-0.5))    //This method of
weight generation was chosen to improve the speed of convergence
            ENDFOR
        ENDIF
    ENDFOR
```

runner:

Implemented in Python, as certain functions are non-obvious in pseudo code.

Algorithm:

```
def runner(nodes_list, number_of_iterations, data_list, learning_coefficient=0.25):
    iterations_completed = 0
    print("[", end="")
    for i in range(number_of_iterations * len(data_list)):
        iterations_completed+=1
        if (100 * (iterations_completed/(number_of_iterations*len(data_list)))) % 10 == 0:
            print("=", end="")
        desired_value = data_list[i % len(data_list)][1]
        for node in nodes_list:
            if node.is_input:
                node.calculate_net(data_list[i % len(data_list)][0][node.row])
            else:
                node.calculate_net()

        for node in reversed(nodes_list):
            if not node.is_input:
                node.change_weights(desired_value, learning_coefficient)
            else:
                break
    print("]",end="\n")
```

Design of Digit Drawing Tool:

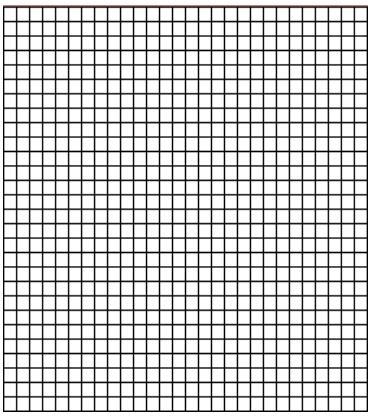
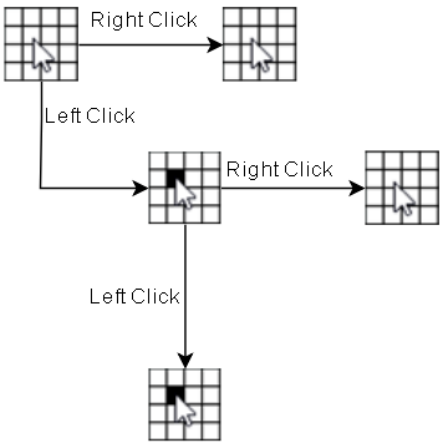


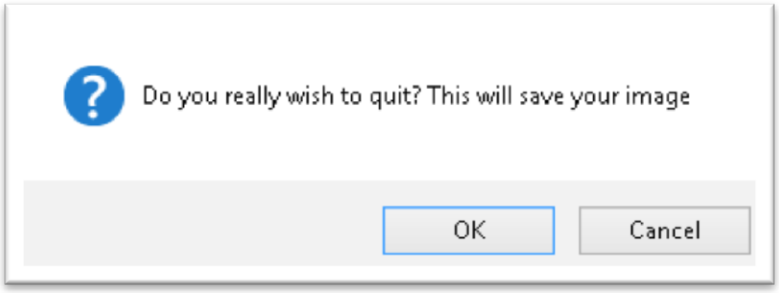
Figure 16 - A 28 by 28 grid for drawing

The design of the digit drawing space should be simple, and easy to use. We simply need a 28 by 28 grid, when the user left clicks a square it should change to black – when they right click it, that particular square should be changed back to white (erased).

When the user has finished drawing, they should close the program, at which point the program should prompt them if they are ready to quit and inform them that their drawing will be saved.



When the user clicks OK, the program should save the image in a default file space and close. Should they click Cancel they should be able to return and continue drawing.

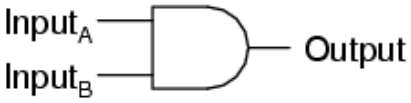


The file should be saved in the same manner as the MNIST data, i.e. in a list 768 values long, normalized with mean zero and standard deviation 1.

Technical Solutions and Explanations:

Explanation of Linear Neural Network Technical Solution

Before we start to define functions we need to decide on a data set. It has to be suitably simple that the system will be able to arrive at a solution, yet complicated enough that a simple guess wouldn't arrive at the correct answer.



A logical object perfect for the task would be the logical definition of an AND gate, however we will test the system on other logic gates.

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

The system of inputs and outputs can be rendered easily in Python as such (The extra 1 in our inputs is for the bias of every node, previously described):

```
data = [([1, -1, -1], -1), ([1, -1, 1], -1), ([1, 1, -1], -1), ([1, 1, 1], 1)]
```

First we need to define a function for calculating net; it needs the weights of the nodes and the data as input.

```
def calculate_net(weights_for_net, data_value):
```

We can start by defining the net value as zero initially

```
net = 0
```

$$net_p = \sum_{i=0}^{n} \left(w_i x_i \right)_p$$

Then we want to iterate through our data values sum the product of them and our weights:

```
for index, data_point in enumerate(data_value[0]):
    net += weights_for_net[index] * data_point
```

And finally we want to return the net value:

```
return net
```

Then we need a function to compute our delta rule, remember:

$$f(net, d) = \delta = d_p - net_p$$

So first we'll define our function:

```
def calculate_delta_rule(desired_data, net_value_for_delta_rule):
```

Where our input is the desired value for a pattern and the net value, then we can simply return the desired data minus the net value:

```
    return desired_data - net_value_for_delta_rule
```

Then we need a function to calculate how much we want our weights to change, remember:

$$\Delta w_i = \frac{\eta}{P} \sum_{p=1}^{P} \delta_p \cdot x_{ip}$$

So our function can be defined as:

```
def get_weight_change(weight_changes, delta_rule_value, supplied_data, offset_value=0.1):
```

Where it's given a list of weights to change (weight_changes), the value of our delta rule (delta_rule_value), our pattern set (supplied_data) and our offset value (offset_value) which will have a value of 0.1 if unedited.

```
for placeholder, data_points in enumerate(supplied_data[0]):
    weight_changes[placeholder] += delta_rule_value * data_points * offset_value
```

Our function should iterate through the supplied data and apply the function and apply our equation, namely: The sum of Delta rule * data point * offset value and record the values in our weight changes list: The enumerate function simply loops through an iterable object and stores the point it is in the list and the value of the list like so:

```
1  colour_list = ["red", "green", "blue" ]
2
3  for placeholder, colour in enumerate(colour_list):
4      print(placeholder, colour)
```

```
0 red
1 green
2 blue

Process finished with exit code 0
```

Our change in weights calculated and stored, we can then output our value:

```
return weight_changes
```

Our last function to produce is one to change our weight values; it should take our weight changes and some weights to change:

```
def change_weights(weight_changes, weights_for_changing):
```

We simply want to iterate through our list of weights (weights_for_changing), and add the corresponding value from weight_changes

```
for counter in range(len(weights_for_changing)):
    weights_for_changing[counter] += weight_changes[counter]
```

Then we may simply return our list of weights:

```
return weights_for_changing
```

This value of weights for changing would be unnecessary if we were to use a global scope with our lists, however local lists are better practice and renders the code more readable.

The first step is simple:

```
for iterations in range(100):
    weight_change = [0, 0, 0]
```

100 is the number of times we want to repeat the learning rule. After a certain number of iteration, weight values will oscillate around a solution and the system cease to make improvements, so a huge value of iterations will simply waste time.

Next we will iterate through our pattern set and simply apply our previously defined functions.

```
for data_values in data:
    net_value = calculate_net(weights, data_values)
    delta_rule = calculate_delta_rule(data_values[1],
                                      net_value)
    weight_change = get_weight_change(weight_change, delta_rule, data_values,
                                      offset)
```

And then to cap our program off we want to find an average value for our weight changes:

```
for i in range(len(weight_change)):
    weight_change[i] = weight_change[i] / len(data)
```

And save our final weight values as our weights:

```
weights = change_weights(weight_change, weights)
```

And our program is complete!
Now to test our program I want to be able to store certain values in a file, to do this I must first create a text file. Luckily Python's file reading and editing module is easy to use and perfect for this situation:

```
f = open("data_storage.txt", "w")
```

Now if I want to write things to our file ("data_storage.txt") I can **simply** use:

```
f.write("{0}, {1}, {2} , {3}\n".format(str(" ".join(['%.2f' % elem for elem in weights])),
                                       str(" ".join(['%.2f' % elem for elem in weight_change])), str(
        round(net_value, 2)), str(data_values[0])))
```

This simple solution is so elegant and easy to read I need not explain it, Pythonic beauty through and through.
The .format() inputs the values of its arguments into the curly bracket marked 0, 1, 2 and 3.
Its first argument:

```
str(" ".join(['%.2f' % elem for elem in weights]))
```

The list comprehension here writes every element in weights to 2

significant figures, stores that to a list and then adds spaces between all the elements of the list. Finally it stores the output as a string and that is displayed.
The second argument:

```
str(" ".join(['%.2f' % elem for elem in weight_change]))
```

Does exactly the same, but this time for our list of weight changes.

Its third arguments are simply rounding values for our net_value, converting that to a string and then displaying the result, and the fourth does nothing more than save the first element of our data_values.
The first iteration can now be displayed with initial weights -0.12, 0.4 & 0.65:

x_0	x_1	x_2	w_0	w_1	w_2	net	d	x_0 *η*δ	x_1 *η*δ	x_2 *η*δ
1	-1	-1	-0.12	0.4	0.65	-1.17	-1	0.02	-0.02	-0.02
1	-1	1	-0.12	0.4	0.65	0.13	-1	-0.1	0.1	-0.13
1	1	-1	-0.12	0.4	0.65	-0.37	-1	-0.16	0.03	-0.07
1	1	1	-0.12	0.4	0.65	0.93	1	-0.15	0.04	-0.06
							Total:	-0.39	0.15	-0.28
							Mean:	-0.1	0.04	-0.07

The data values of an AND gate are initialised, the net is calculated and finally the mean change in weights is found and implemented in the next iteration:

x_0	x_1	x_2	w_0	w_1	w_2	net	d	x_0 *η*δ	x_1 *η*δ	x_2 *η*δ
1	-1	-1	-0.16	0.41	0.64	-1.2	-1	0.02	-0.02	-0.02
1	-1	1	-0.16	0.41	0.64	0.07	-1	-0.09	0.09	-0.13
1	1	-1	-0.16	0.41	0.64	-0.38	-1	-0.15	0.02	-0.07
1	1	1	-0.16	0.41	0.64	0.89	1	-0.14	0.04	-0.05
							Total:	-0.36	0.13	-0.27
							Mean:	-0.09	0.03	-0.07

The current net value for input values of 1, 1, and -1 is wrong by quite a

lot (outputting 0.07 rather than -1); however the other values are quite close to our desired values.
After some 42 iterations the program starts to repeat itself and oscillates around the weights -0.5, 0.5 and 0.5:

x_0	x_1	x_2	w_0	w_1	w_2	net	d	x_0 *η*δ	x_1 *η*δ	x_2 *η*δ
1	-1	-1	-0.5	0.5	0.5	-1.5	-1	0.05	-0.05	-0.05
1	-1	1	-0.5	0.5	0.5	-0.49	-1	0	0	-0.1
1	1	-1	-0.5	0.5	0.5	-0.5	-1	-0.05	-0.05	-0.05
1	1	1	-0.5	0.5	0.5	0.51	1	0	0	0
							Total:	0	-0.1	-0.2
							Mean:	0	-0.03	-0.05

The next iteration is practically the same:

x_0	x_1	x_2	w_0	w_1	w_2	net	d	x_0 *η*δ	x_1 *η*δ	x_2 *η*δ
1	-1	-1	-0.5	0.5	0.5	-1.5	-1	0.05	-0.05	-0.05
1	-1	1	-0.5	0.5	0.5	-0.49	-1	0	0	-0.1
1	1	-1	-0.5	0.5	0.5	-0.5	-1	-0.05	-0.05	-0.05
1	1	1	-0.5	0.5	0.5	0.5	1	0	0	0
							Total:	0	-0.1	-0.2
							Mean:	0	-0.03	-0.05

The outputs of the system for different inputs are as such:

x_0	x_1	x_2	net
1	-1	-1	-1.5
1	-1	1	-0.5
1	1	-1	-0.5
1	1	1	0.5

Which when rounded garners:

x_0	x_1	x_2	net

1	-1	-1	-1
1	-1	1	-1
1	1	-1	-1
1	1	1	1

The net values therefore exactly mirror our desired value, the program has succeeded in determining a set of weights to solve an AND gate!

```
data = [([1, -1, -1], -1), ([1, -1, 1], 1), ([1, 1, -1], 1), ([1, 1, 1], 1)]
```

To test an inclusive OR gate say, all we need do is edit the data values to mirror that of such a gate, i.e.:
That is:

Input A	Input B	Output
-1	-1	0
-1	1	1
1	-1	1
1	1	1

With the same set of initial weights our program begins as such:

x_0	x_1	x_2	w_0	w_1	w_2	net	d	x_0 *η*δ	x_1 *η*δ	x_2 *η*δ
1	-1	-1	-0.12	0.4	0.65	-1.17	-1	0.02	-0.02	-0.02
1	-1	1	-0.12	0.4	0.65	0.13	1	0.1	-0.1	0.07
1	1	-1	-0.12	0.4	0.65	-0.37	1	0.24	0.03	-0.07
1	1	1	-0.12	0.4	0.65	0.93	1	0.25	0.04	-0.06
							Total:	0.61	-0.05	-0.08
							Mean:	0.15	-0.01	-0.02

x_0	x_1	x_2	w_0	w_1	w_2	net	d	x_0 *η*δ	x_1 *η*δ	x_2 *η*δ
1	-1	-1	-0.06	0.41	0.64	-1.11	-1	0.01	-0.01	-0.01
1	-1	1	-0.06	0.41	0.64	0.17	1	0.09	-0.09	0.07
1	1	-1	-0.06	0.41	0.64	-0.29	1	0.22	0.03	-0.06
1	1	1	-0.06	0.41	0.64	0.99	1	0.22	0.04	-0.05
							Total:	0.54	-0.03	-0.05
							Mean:	0.14	-0.01	-0.01

x_0	x_1	x_2	w_0	w_1	w_2	net	d	x_0 *η*δ	x_1 *η*δ	x_2 *η*δ
1	-1	-1	0	0.42	0.62	-1.04	-1	0	0	0
1	-1	1	0	0.42	0.62	0.2	1	0.08	-0.08	0.08
1	1	-1	0	0.42	0.62	-0.2	1	0.2	0.04	-0.04
1	1	1	0	0.42	0.62	1.04	1	0.2	0.03	-0.05
							Total:	0.48	-0.01	-0.01
							Mean:	0.12	0	0

x_0	x_1	x_2	w_0	w_1	w_2	net	d	x_0 *η*δ	x_1 *η*δ	x_2 *η*δ
1	-1	-1	0.05	0.43	0.61	-0.99	-1	0	0	0
1	-1	1	0.05	0.43	0.61	0.23	1	0.08	-0.08	0.08
1	1	-1	0.05	0.43	0.61	-0.13	1	0.19	0.04	-0.04
1	1	1	0.05	0.43	0.61	1.09	1	0.18	0.03	-0.04
							Total:	0.45	-0.01	0
							Mean:	0.11	0	0

After 47 iterations this time the system arrives at a solution (Starts to repeat itself):

x_0	x_1	x_2	w_0	w_1	w_2	net	d	x_0 *η*δ	x_1 *η*δ	x_2 *η*δ
1	-1	-1	0.5	0.5	0.5	-0.5	-1	-0.05	0.05	0.05
1	-1	1	0.5	0.5	0.5	0.5	1	0	0	0.1
1	1	-1	0.5	0.5	0.5	0.5	1	0.05	0.05	0.05
1	1	1	0.5	0.5	0.5	1.5	1	0	0	0
							Total:	0	0.1	0.2
							Mean:	0	0.03	0.05

x_0	x_1	x_2	w_0	w_1	w_2	net	d	x_0 *η*δ	x_1 *η*δ	x_2 *η*δ
1	-1	-1	0.5	0.5	0.5	-0.5	-1	-0.05	0.05	0.05
1	-1	1	0.5	0.5	0.5	0.5	1	0	0	0.1
1	1	-1	0.5	0.5	0.5	0.5	1	0.05	0.05	0.05
1	1	1	0.5	0.5	0.5	1.5	1	0	0	0
							Total:	0	0.1	0.2
							Mean:	0	0.03	0.05

And again our rounded values for the net mirror our desired values:

x_1	x_2	net
-1	-1	-1
-1	1	1
1	-1	1
1	1	1

And our system has arrived at a solution.
However it is worth noting that the system will never rest on exactly 0.5, 0.5, 0.5 it will always be some percentage off, even when the correct solution has been found. This is because the weighted sum will always deviate slightly from our desired values.
This does mean, however, that if another data value or class of data value is added, the weights should change with “minimum disturbance” i.e. the system will attempt to accommodate for the new input class while preserving its ability to resolve previous input patterns.

Solution for Linear Neural Network:

```
weights = [-0.12, 0.4, 0.65]
data = [([1, -1, -1], -1), ([1, -1, 1], -1), ([1, 1, -1],
1), ([1, 1, 1], -1)]

offset = 0.1
f = open("TESTING.txt", "w")

def calculate_net(weights_for_net, data_value):
    net = 0
    for index, data_point in enumerate(data_value[0]):
        net += weights_for_net[index] * data_point
    return net


def calculate_delta_rule(desired_data,
net_value_for_delta_rule):
    return (desired_data - net_value_for_delta_rule)
```

```
def change_weights(weight_changes, weights_for_changing):
    for index in range(len(weights_for_changing)):
        weights_for_changing[index] +=
weight_changes[index]


def get_weight_change(delta_rule_value, supplied_data,
offset_value=0.1):
    for placeholder, data_points in
enumerate(supplied_data[0]):
        weight_change[placeholder] += delta_rule_value *
data_points * offset_value
    return weight_change


for iterations in range(1000):
    weight_change = [0, 0, 0]
    for data_values in data:
        net_value = calculate_net(weights, data_values)
        delta_rule = calculate_delta_rule(data_values[1],
net_value)

        weight_change = get_weight_change(delta_rule,
data_values, offset)

        f.write("{0}, {1}, {2}, {3}\n".format(str("
".join(['"%.2f' % elem for elem in weights])), str("
".join(['%.2f' % elem for elem in weight_change])),
str(round(net_value, 2)), str(data_values[1])))

    weight_change = [x/len(data) for x in weight_change]
    change_weights(weight_change, weights)

for i in range(4):
    print(round(calculate_net(weights, data[i]), 1))

f.close()
```

Drawing Tool Technical Solution Explanation:

The Drawing Tool was implemented exactly as decided in the design stage, to do this I used the Python built in package Tkinter.

First I made an "App" class that would include the necessary methods to create and display images. I had to inherit the methods of the Tkinter class:

```
class App(tk.Tk):

    def __init__(self):
        tk.Tk.__init__(self)
```

Then I created a Tkinter object called a canvas and "packed it":

```
self.canvas = tk.Canvas(self, width=280, height=280, borderwidth=0, highlightthickness=0)
self.canvas.pack()
```

The canvas object is a Tkinter object which "provides structured graphics facilities for Tkinter. This is a highly versatile widget which can be used to draw graphs and plots, create graphics editors, and implement various kinds of custom widgets."[13] The variable highlight thickness controls the thickness of the grey border that usually surrounds this canvas, making it equal zero removes this grey border.

The pack object simply creates the canvas in our Tkinter window.

I then defined some attributes for our class, including a dictionary that would hold all the pixels of our image.

```
self.rows = 28
self.columns = 28
self.cellwidth = 10
self.cellheight = 10

self.rect = {}
```

I then created our grid of 28 by 28 squares.

```
for column in range(28):
    for row in range(28):
        x1 = column*self.cellwidth
        y1 = row * self.cellheight
        x2 = x1 + self.cellwidth
        y2 = y1 + self.cellheight
        self.rect[row,column] = self.canvas.create_rectangle(x1,y1,x2,y2, fill="white", tags="rect")
```

This starts by finding x1 and y1 which are the coordinates for the top left corner of any one rectangle being drawn. Then the coordinates of the

[13] http://effbot.org/tag/Tkinter.Canvas

bottom right corner are found by adding the width and height of the rectangle to its top left corner coordinates.

Finally I created a Tkinter object and stored it in our dictionary, each square's key is its place on the grid (its row and column).

```
self.canvas.bind("<Button-1>", get_square)
self.canvas.bind("<B1-Motion>", get_square)
self.canvas.bind("<Button-3>", get_square)
self.canvas.bind("<B3-Motion>", get_square)
```

I then bound left click and right click to a function 'get_square', this function will find the square being clicked. When a lick occurs on the grid a Tkinter event will be passed to 'get_square', this event can be used to find where the mouse has clicked and thus which grid.

Button 1 is left click & Button 3 is right click, and the Motion tag refers to when the button is held down. So with motion the user can draw continuous lines (or erase in a continuous line) rather than having to click each square individually.

```
def get_square(event):
    row_clicked = math.floor(event.x/10)
    column_clicked = math.floor(event.y/10)
```

First we find the grid clicked, event.x and event.y refers to which pixel in the Tkinter canvas has been clicked. As every square in our grid is 10 by 10 pixels, we first divide the event.x and event.y by 10, the math.floor simply rounds the value down.

Then we find the corresponding square by using the key on our dictionary:

```
def get_square(event):
    row_clicked = math.floor(event.x/10)
    column_clicked = math.floor(event.y/10)
    item_id = self.rect[column_clicked, row_clicked]
```

The right click event has an "event number" of 3 while holding right click has an "event state" of 1032. These are simply values Tkinter applies to these two events. So we can check for those, and in the event of a right click we want the colour of our square to be changed to white:

```
def get_square(event):
    row_clicked = math.floor(event.x/10)
    column_clicked = math.floor(event.y/10)
    item_id = self.rect[column_clicked, row_clicked]
    if event.num == 3 or event.state == 1032:
        paint_square(item_id, colour="white")
    else:
        paint_square(item_id)
```

And then finally we want our paint square function:

```
def paint_square(item_id, colour="black"):
    self.canvas.itemconfig(item_id, fill=colour)
```

We pass item_id to the paint square function, so we know which square to change the colour of.

Now we have our grid, which we can left click to draw on and right click to erase:

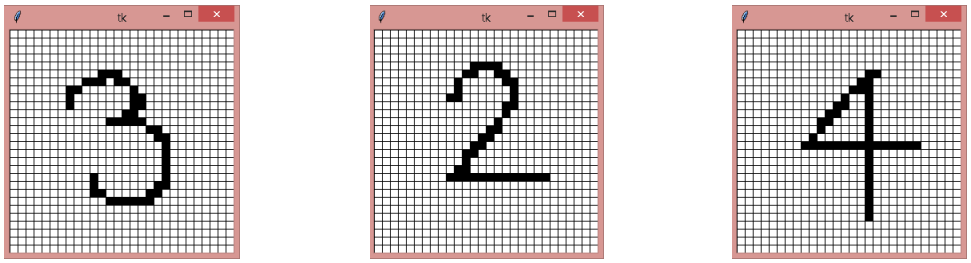


Figure 17 - Example drawings of Digits in the tool

However in using the tool, I found that using the erase tool could sometimes be cumbersome so I created a function to completely clear the canvas and bound it to ‘c’:

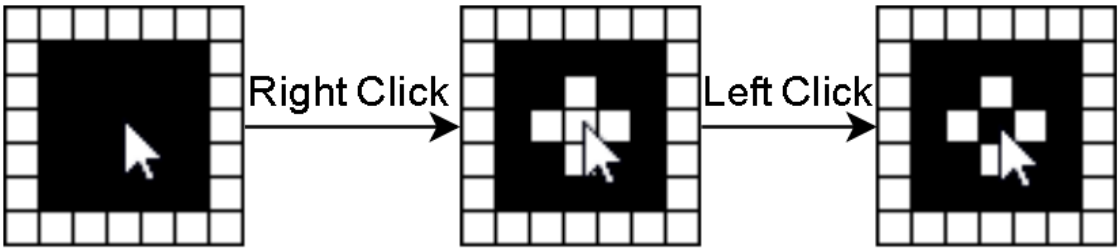
```
self.canvas.bind("<c>", clear_canvas)

def clear_canvas(event):
    print("Canvas Cleared")
    self.canvas.itemconfig('rect', fill="white")
```

Where itemconfig is an inherited Tkinter method, inherited earlier.

To do this I had to set the ‘focus’ to the canvas, this is because key bindings only fire when the widget with the keyboard focus gets a key event:

```
app.canvas.focus_set()
```

I also decided to increase the size of the eraser:

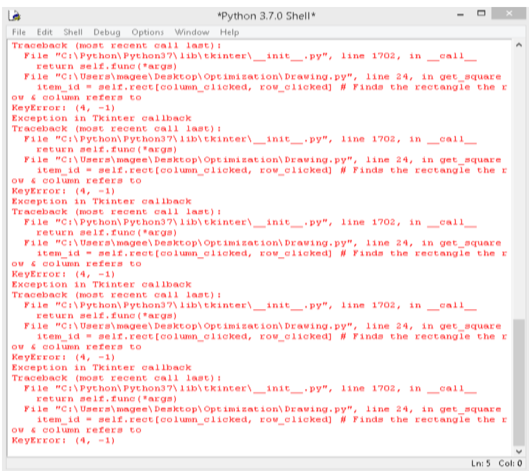


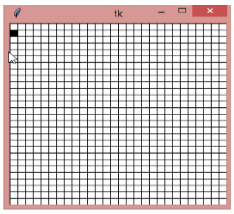
This was done by first checking that the cursor wasn't at the edge of the grid (To make sure the program isn't trying to erase any non-existent squares off-screen):

```
if event.num == 3 or event.state == 1032:
    if 0 < row_clicked < 28 and 0 < column_clicked < 28:
        paint_square(self.rect[column_clicked+1, row_clicked], colour="white")
        paint_square(self.rect[column_clicked-1, row_clicked], colour="white")
        paint_square(self.rect[column_clicked,  row_clicked+1], colour="white")
        paint_square(self.rect[column_clicked,  row_clicked-1], colour="white")
    paint_square(item_id, colour="white")
```

And then painting squares white around the cursor. Note that even if the cursor is at the edge of the grid, whichever square it clicks will be erased however you won't get the extra 4 squares erased.

The next problem that arose was that whenever the cursor clicked at the edge of the grid, a red error message would show up. The program wouldn't close but it looked ugly and pulls attention from the drawing:





This was because the cursor was clicking off-screen, and the event was passing negative values for its coordinates.

To fix this I simply set a bound to where clicks would be accepted from:

```
def get_square(event):
    if 0 <= event.x < 280 and 0 <= event.y < 280:
```

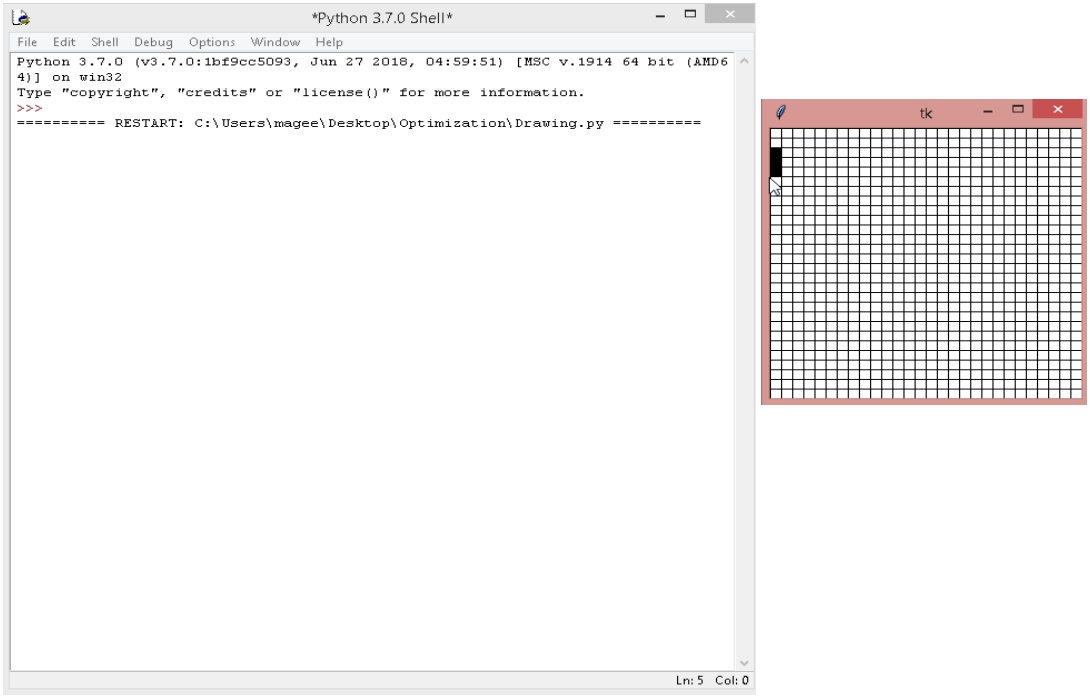


Figure 18 - Error Fixed

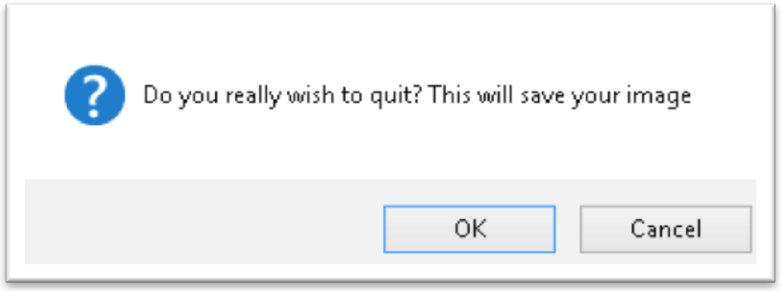
To create our save and quit message box, I had to import message box from Tkinter and first write:

```
app.protocol("WM_DELETE_WINDOW", save_and_quit)
```

This Tkinter method passes the burden of closing the window to a function of your own creation. When the user clicks close button at the top right of the window, my save_and_quit function is run.

The function first creates a message box and checks if the user agreed to close:

```
if tk.messagebox.askokcancel("Quit", "Do you really wish to quit? This will save your image"):
```



If the user clicks Cancel, the window simply closes and they may continue drawing. Otherwise we iterate through our dictionary of squares and save any square coloured white as a 0 and any square coloured black a 1:

```
output = []
for square in app.rect:
    if (app.canvas.itemcget(app.rect[(square[1], square[0])], "fill")) == "white":
        output.append(0)
    else:
        output.append(1)
```

Here itemcget finds the colour of the squares.

Finally we open a file named ‘drawing.txt’ and write our list of values (without the opening and closing brackets) to that file, and then close the file and shuts down the Tkinter window:

```
file = open("drawing.txt", "w")
file.write(str(output)[1:-1])
file.close()
app.destroy()
```

And finally I decided to have the left click create the same shape as the right, it made drawing easier and is more akin to the digits in the MNIST dataset.

<u>Drawing Tool Technical Solution:</u>

```
import tkinter as tk
from tkinter import messagebox
import random
import math
import inspect

class App(tk.Tk):

    def __init__(self):
        tk.Tk.__init__(self)   # inherits tkinter class
        self.canvas = tk.Canvas(self, width=280, height=280, borderwidth=0,
highlightthickness=0)   # creates a tkinter canvas
        self.canvas.pack()
        self.rows = 28
        self.columns = 28
        self.cellwidth = 10
        self.cellheight = 10

        self.rect = {}   # creates a dictionary of all our squares

        def get_square(event):
            if 0 <= event.x < 280 and 0 <= event.y < 280:   #if the cursor has
```

```
clicked a spot in the grid
                row_clicked = math.floor(event.x/10)   # finds the row of the
rectangle clicked
                column_clicked = math.floor(event.y/10)   # finds the column of
the rectangle clicked
                item_id = self.rect[column_clicked, row_clicked] # finds the
rectangle the row & column refers to
                if event.num == 3 or event.state == 1032:   # if the event is a
single right click or a dragged right click
                    if 0 < row_clicked < 27 and 0 < column_clicked < 27:   # if
the cursor is not on the edge, erase in a larger shape
                        paint_square(self.rect[column_clicked+1, row_clicked],
colour="white")
                        paint_square(self.rect[column_clicked-1, row_clicked],
colour="white")
                        paint_square(self.rect[column_clicked,  row_clicked+1],
colour="white")
                        paint_square(self.rect[column_clicked,  row_clicked-1],
colour="white")
                    paint_square(item_id, colour="white")
                else:
                    paint_square(item_id)
                    if 0 < row_clicked < 27 and 0 < column_clicked < 27:
                        paint_square(self.rect[column_clicked+1, row_clicked],
colour="black")
                        paint_square(self.rect[column_clicked-1, row_clicked],
colour="black")
                        paint_square(self.rect[column_clicked,  row_clicked+1],
colour="black")
                        paint_square(self.rect[column_clicked,  row_clicked-1],
colour="black")

        def paint_square(item_id, colour="black"):
            self.canvas.itemconfig(item_id, fill=colour)   # Change the colour of
whatever square is necessary

        def clear_canvas(event):
            print("Canvas Cleared")
            self.canvas.itemconfig('rect', fill="white")    # Changes the colour
of every square on screen

        self.canvas.bind("<Button-1>", get_square)
        self.canvas.bind("<B1-Motion>", get_square)
        self.canvas.bind("<Button-3>", get_square)
        self.canvas.bind("<B3-Motion>", get_square)
        self.canvas.bind("<c>", clear_canvas)

        for column in range(28):
            for row in range(28):
                x1 = column*self.cellwidth # Defines the x value of the top-left
corner of our rectangle
                y1 = row * self.cellheight # Likewise for the y value
                x2 = x1 + self.cellwidth # Finds the x value for the bottom-left
corner of our rectangle
                y2 = y1 + self.cellheight # Likewise for the y value
                self.rect[row,column] = self.canvas.create_rectangle(x1,y1,x2,y2,
fill="white", tags="rect") # Creates a tkinter rectangle object
```

```
if __name__ == "__main__":

    def save_and_quit():
        if tk.messagebox.askokcancel("Quit", "Do you really wish to quit? This
will save your image"):
            output = []
            for square in app.rect:   # iterates through all rectangles
                if (app.canvas.itemcget(app.rect[(square[1], square[0])], "fill"))
== "white":   # checks if the square is white
                    output.append(0)
                else:
                    output.append(1)   # if the square isnt black, append 1
            file = open("drawing.txt", "w")    # opens a file
            file.write(str(output)[1:-1])   # saves our list to that file
            file.close()
            app.destroy()   # closes our tkinter window

    app = App()
    app.canvas.focus_set()   # ensures the canvas detects key presses
    app.protocol("WM_DELETE_WINDOW", save_and_quit)
    app.mainloop()
```

Backpropagation Technical Solution Explanation

The first thing to do was create a Python class for our node object, in Python that is simply done:

```
class Node:
    def __init__(self, row, layer, is_input=False, is_output=False):

        self.connections = []  # Stores the nodes this node outputs to
        self.inputs = []  # Stores the nodes this node receives input from
        self.row = row  # Defines the node's row
        self.layer = layer  # Defines the node's layer
        self.is_input = is_input  # Defines if the node is an input node
        self.is_output = is_output  # Defines if the node is an output node
        self.weights = []  # Stores the node's weights
        self.old_weights = []  # Stores the node's weights before they're changes, for use in backpropagation.
        self.net = 0  # Stores the node's net value
        self.output_value = 0  # Stores the node's output value
        self.change_in_weight = 0  # Stores the node's change in weight value
        self.delta_rule = 0  # Stores the node's delta rule
```

Then I defined a function network creator, it takes a list containing the structure of our network and instantiates the necessary nodes:

```
nodes = [2,2,1]
nodes = [10, 100, 1]
nodes = [784, 150, 10]
```

Figure 19 - Example network structures

```
def network_creator(node_list, structure_list):
    for layer, number_of_nodes in enumerate(structure_list):
        if layer == 0:
            for row in range(number_of_nodes):
                node_list.append(Node(row, layer, is_input=True))
        elif layer == len(structure_list) - 1:
            for row in range(number_of_nodes):
                node_list.append(Node(row, layer, is_input=False, is_output=True))
        else:
            for row in range(number_of_nodes):
                node_list.append(Node(row, layer))
```

If the node is in the first layer (and is therefore an input node) network creator instantiates it with is_input set to true, alternatively if the node is in the final layer it's instantiated with is_output set to true.

We then run connect_all_nodes to do exactly that:

```
def connect_all_nodes(nodes):
    for i in nodes:
        if not i.is_output:
            for j in nodes:
                if j.layer == i.layer + 1:
                    i.add_connections(j.row, j.layer, nodes)
```

The function iterates through all the nodes twice and connects every node in one layer to every node in the next. The problem I found with this function is its very slow speed. Its time complexity is quadratic, and for large networks it was taking a ridiculous amount of time.

So I rewrote it, using Python list comprehensions:

```
def connect_all_nodes(nodes, network_structure):
    for layer_no, layer in enumerate(network_structure[:-1]):
        for output_node in nodes[0:layer]:
            node_up_to = sum(network_structure[0:(layer_no + 1)])
            for input_node in nodes[node_up_to:node_up_to + network_structure[layer_no + 1]]:
                output_node.add_connections(input_node.row, input_node.layer, nodes, network_structure)
```

Improvements:

- Doesn't iterate through the final layer, nodes in the output layer have no layer following them and have already been connected to the layer preceding them.
- Doesn't iterate through nodes in its layer and check whether they are in their layer, using the network structure list it can iterate only though nodes in the next layer.
- Doesn't have to check if it's connecting to itself or a node in its layer **or** any further layers.
- Variable naming makes it easier to understand what's going on.

The Node method "add_connections" is simply:

```
def add_connections(self, desired_row, desired_layer, node_list, network_structure):
    if not (desired_row == self.row and desired_layer == self.layer):
        node_id = sum(network_structure[0:desired_layer]) + desired_row
        desired_node = node_list[node_id]

        if desired_node.row == desired_row and desired_node.layer == desired_layer:
            self.connections.append(desired_node)
            desired_node.inputs.append(self)
            return None
```

It does some basic error checking to ensure the data is correct, and then adds the desired node to the node that called the methods list of connections. It also adds the node that called the method to the desired node's list of inputs.

Next, I implemented a way for users to load their own network:

```
if get_network:  # If the user wants to load a already created network (variable defined at runtime)
    while True:
        try:
            network_structure, desired_data, layers = get_file_data(input(
                "Please enter the name of the file to be used as input (Without .txt )\n --> "))  # Try and find it
            set_weights(nodes, layers)  # Try and initialize the weights
        except:
            user_input = input("An error has occured, would you like to reenter the name Y or N \n--> ")  # Else load default data
            if user_input.lower() in ["y", "true", "yes", "sure", "i would", "okay", "es"]:
                continue
            else:
                generate_weights(nodes)
                break
```

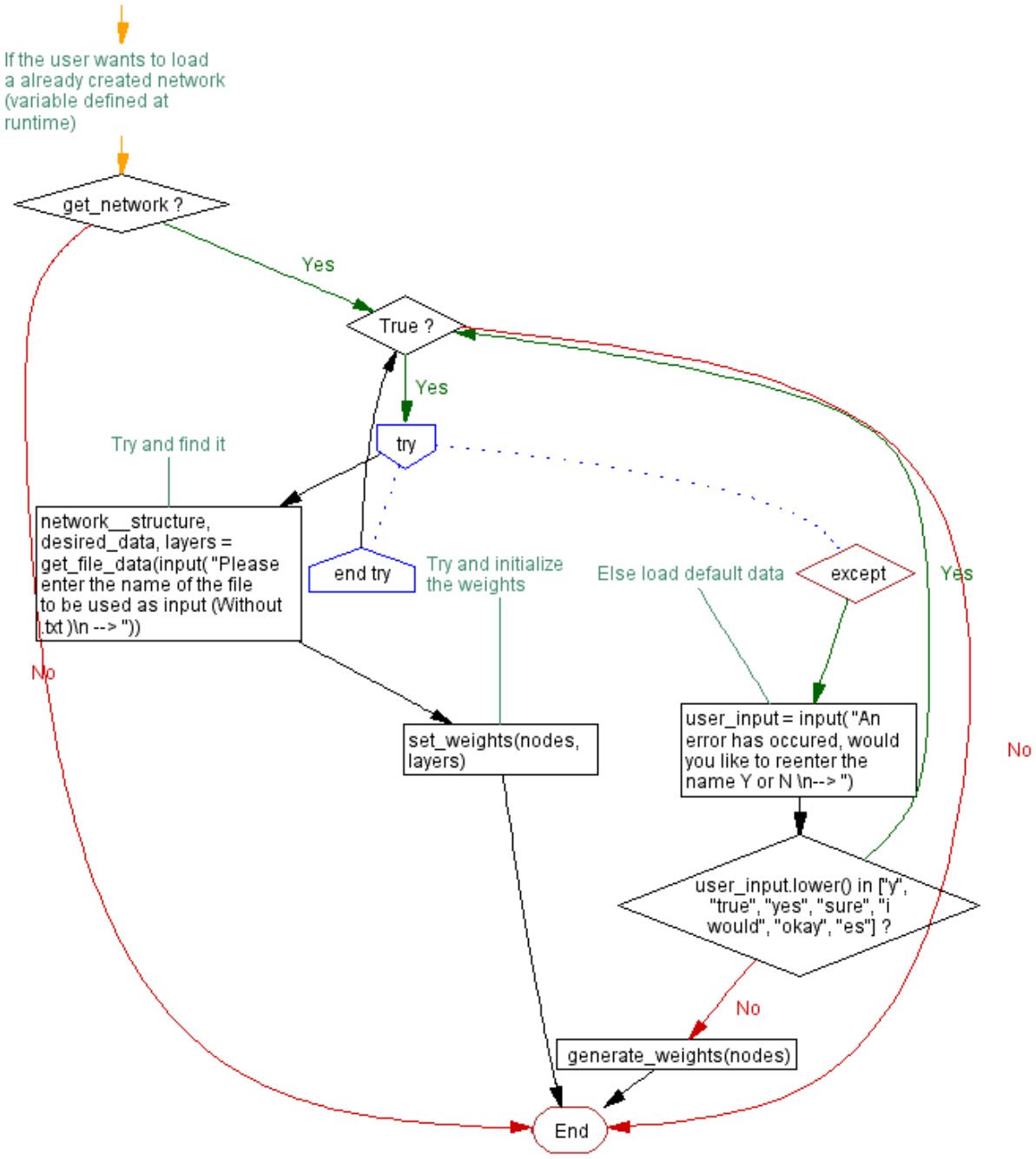


Figure 20 - Flow chart for get_network()

This code simply gets the user's file name, if correct it runs set_weights on those loaded files otherwise it generates weights the normal way.

```
def get_file_data(input_file_name):
    file = open(input_file_name + ".txt", "r")
    file_line_list = file.read().split("\n")
    desired_data = eval(file_line_list[0])
    layers = []
    network_structure = [len(desired_data[0][0])]

    for line in file_line_list[1:-1]:
        layers.append(line.split(":"))

    for layer in layers:
        network_structure.append(len(layer))
        for node_no, node in enumerate(layer):
            node = layer[node_no] = eval(node)
            for weight_number, weight in enumerate(node):
                node[weight_number] = float(weight)

    return network_structure, desired_data, layers
```

'Get file data' opens the requested file, reads it, normalises it and finally serves it back to the main code.

And 'set weights' simply iterates through the list of nodes and sets the weights to those provided (given the node is not an input node).

```
def set_weights(nodes, weight_data):
    for node in nodes:
        if node.layer != 0:
            node.weights = weight_data[node.layer - 1][node.row]
```

I then created a function to get user input on the number of iterations that are to be computed:

```
    iterations = 10  # Default no of iterations
    iterations = get_no_of_iterations(iterations)  # Get user input
```

This simply sets a default number of iterations and then calls the function get_no_of_iterations:

```
def get_no_of_iterations(iterations, nodes_list, data_list):
    print("Default Iterations = " + str(iterations))
    user_input = input("How many Iterations would you like, press enter for default.\n --> ").lower()
    while True:
        try:
            user_input = int(user_input)
            if user_input > 0:
                print("Running " + str(user_input) + " iterations")
                return user_input
            else:
                show(nodes_list, data_list)
                user_input = input("How many Iterations would you like, press enter for default.\n --> ")
        except:
            print("Running " + str(iterations) + " iterations")
            return iterations
```

This first tells the user what the default number of iterations is set to, then asks how many iterations the user would like. If the user enters an integer larger than zero it will return that integer to the main code block and that number of iterations will be ran, if they enter zero the output of the network is shown.

The show function gives the output of the neural network:

```
[784, 300, 10]
Please enter the name of the file to be used as input (Without .txt )
 -->
An error has occured, would you like to reenter the name Y or N
-->
Default Iterations = 10
How many Iterations would you like, press enter for default.
 -->
Running 10 iterations
[0.0 Percent Complete
10.0 Percent Complete
20.0 Percent Complete
30.0 Percent Complete
40.0 Percent Complete
50.0 Percent Complete
60.0 Percent Complete
70.0 Percent Complete
80.0 Percent Complete
90.0 Percent Complete
100 Percent Complete
]
-0.999993912937187 5
-0.9999814825650449 5
-0.999969090727467 5
-0.999990887737938 5
-0.9999999806269617 5
1.5653001666926296 5
-0.9999438260879365 5
-0.9999273899950903 5
-0.9999913672905312 5
-0.9998271421494467 5
The correct node ouputted 1.5653001666926296 and the model predicted 5 which was the correct value
0.99999602821273 0
-0.9999883245007497 0
-0.9999810733650356 0
-0.9999941540693096 0
-0.9999999779435631 0
-0.9998916677639286 0
-0.9999602119336019 0
-0.9999489689090744 0
-0.9999942449891169 0
-0.9934751914066511 0
The correct node ouputted 0.99999602821273 and the model predicted 0 which was the correct value
-0.9999805614964438 4
-0.9999479289819738 4
-0.9999224104782353 4
-0.9999726521581765 4
0.9999997103653032 4
-0.9886362977151597 4
-0.9997620870030629 4
-0.9997002898216987 4
-0.9999702336771465 4
-0.02756402300638603 4
The correct node ouputted 0.9999997103653032 and the model predicted 4 which was the correct value
We had 3 good guesses or a 100.0% success rate
Time taken: 9.130794763565063
Would you like to test a value on this model? Y or N
 -->
```

Figure 21 - Example output from the 'show' function

```
def show(nodes_list, data_list):
    final_layer=nodes_list[len(nodes_list)-1].layer   # gets the final layer of the neural network
    good_guesses = 0
    for placeholder in range(len(data_list)):   # iterates through all the values in the list
        desired_value = data_list[placeholder % len(data_list)][1]   # gets the desired output for the values in the list
        for node in nodes_list:   # propagates forward to calculate error at output
            if node.is_input:
                node.calculate_net(data_list[placeholder % len(data_list)][0][node.row])   # gives the input nodes their input
            else:
                node.calculate_net()

        for node in nodes_list[-10:]:   # iterates through output nodes
            last_layer_outputs = [node.output_value for node in nodes_list[-10:]]    # gets the outputs of all the output nodes
            if node.layer == final_layer:   # if the node is an output node
                print(node.output_value, desired_value)   # show the user it's output
                if node.row == desired_value:
                    correct_node_output = node.output_value   # show the output of the node we want to be firing
                    if node.output_value == max(last_layer_outputs):
                        good_guesses += 1   # if the right node fired, add to good_guesses

        highest_output_index = last_layer_outputs.index(max(last_layer_outputs))   # the index of the node with the highest output
        print("The correct node ouputted",correct_node_output, "and the model predicted", highest_output_index, end=" ")
        if highest_output_index == desired_value:
            print("which was the correct value", end="\n")
        else:
            print("which was the incorrect value", end="\n")

    print("We had", good_guesses, "good guesses or a " + str(good_guesses / (len(data_list)) * 100) + "% success rate")
    if good_guesses / (len(data_list)) > 0.8:
        return True
```

It also informs the user:

- The output of every neuron
- Which neuron fired
- Whether that neuron was the correct one
- What the correct neuron gave as output
- The total number of correct guesses and the percentage of correct guesses.
- What percentage of iterations have been computed (a loading bar)
- How long the model was learning for
- The digit the model was attempting to classify

```
[0.0 Percent Complete
10.0 Percent Complete
20.0 Percent Complete
30.0 Percent Complete
40.0 Percent Complete
50.0 Percent Complete
```

Figure 22 - Loading bar filling up

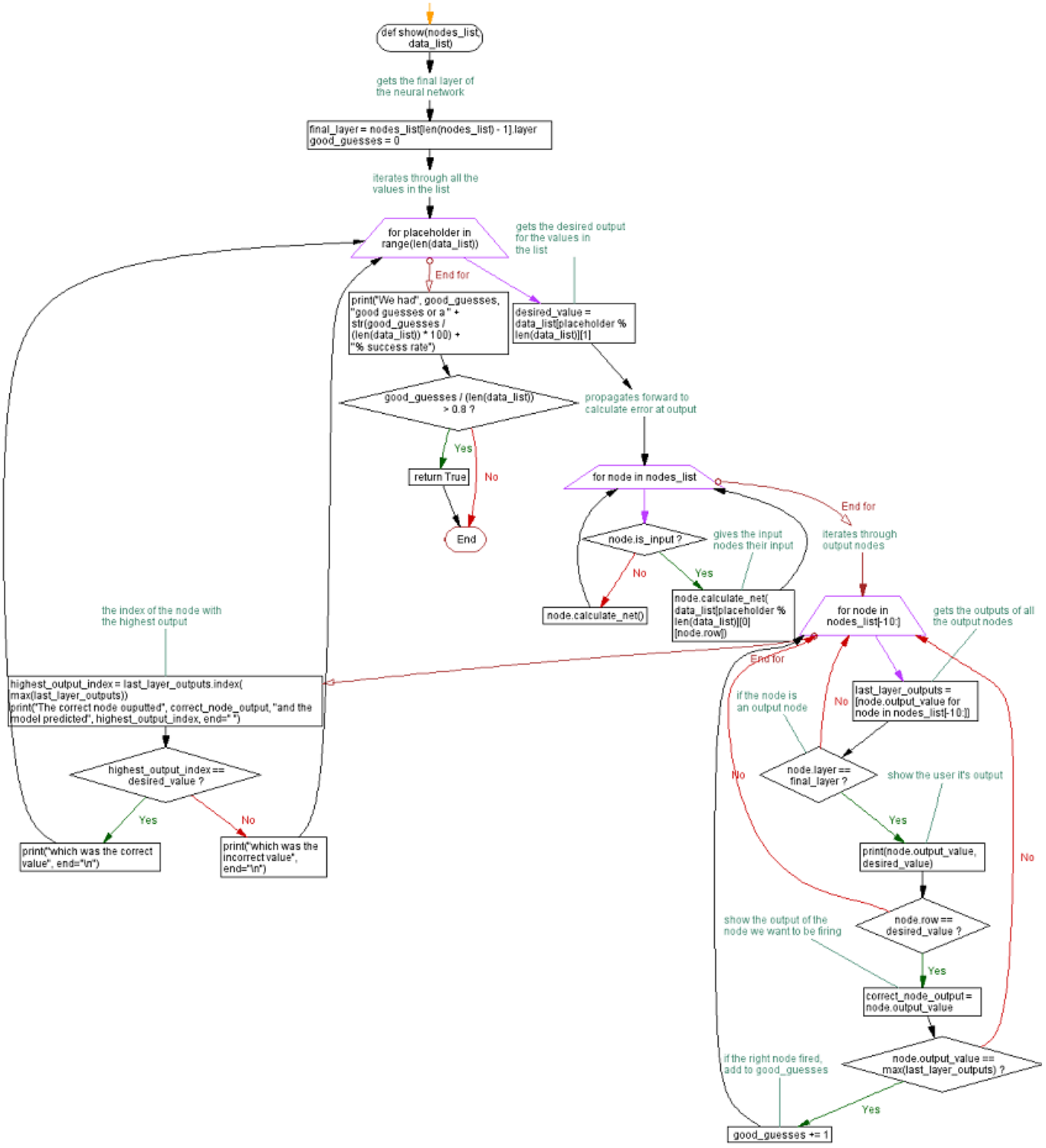


Figure 23 - Flowchart for Show()

Next I implemented a method to test values on our model, the test data will come from the drawing tool

```
while input("Would you like to test a value on this model? Y or N\n --> ").lower() in ["y", "true", "yes", "sure",
                                                                                       "i would", "okay", "es"]:
    if not get_user_input(nodes):   # if the get_user_input function returns false
        break   # exit the loop
```

This code block then calls “get_user_input” with the nodes list as input.

```
def get_user_input(nodes):
    file_name = input("Please enter the desired name of your file. (Without .txt)\n --> ")

    while (file_name + ".txt") not in os.listdir():   # searches the directory for the file the user entered
        #   if the file isn't found, ask the user for input again
        user_input = input("Sorry that file isn't in this directory. Would you like to try another file? Y or N\n --> ")
        if user_input.lower() in ["y", "true", "yes", "sure", "i would", "okay", "es"]:   # if they would
            file_name = input("Please enter the desired name of your file. (Without .txt)\n --> ")   # try again
        else:
            return False   # otherwise return false

    try:   # catch any errors
        data_file = open(file_name + ".txt", "r")   # open their file
        file_data = data_file.read().split(",")   # split the file according to commas
                                                  # this is how data from the drawing tool is formatted
        file_data = list(map(float, file_data))   # converts all the data to floats
        mean = statistics.mean(file_data)    # gets the mean of the input data
        std_dev = statistics.stdev(file_data)   # gets the standard deviation of the input data
        for placeholder, data_point in enumerate(file_data):
            file_data[placeholder] = (data_point - mean) / std_dev   # normalizes the data
    except:
        print("That file type isn't supported, data must be seperated by commas")
        return None

    show_output_from_input(nodes, file_data)
```

This function reads the input, and then normalizes for faster convergence. If the user decides not to supply input it returns false and the program continues.

The main bulk of the program occurs in the function “runner”:

```
def runner(nodes_list, number_of_iterations, data_list, learning_coefficient=0.25):
    iterations_completed = 0
    print("[", end="")   # Starts the loading bar
    for i in range(number_of_iterations * len(data_list)):   # iterates through all the values in the data list
        if i % len(data_list) == 0:
            print(100*round(i/(number_of_iterations * len(data_list)), 2), "Percent Complete")   # gives the percentage completed
            learning_coefficient = learning_coefficient * 0.9   # decreases the learning coefficient for faster convergence
            iterations_completed += 1
        desired_value = data_list[i % len(data_list)][1]   # loads the desired values (the desired values is always the second
                                                           # value in a data point) i.e: [data, label]
        for node in nodes_list:   # iterates through all the nodes
            if node.is_input:
                node.calculate_net(data_list[i % len(data_list)][0][node.row])   # supplies input nodes with their data
            else:
                node.calculate_net()   # supplies non-input node with no data

        for node in reversed(nodes_list):   # propagating backwards through the list of nodes (as defined by backpropagation formula)
            if not node.is_input:
                node.change_weights(desired_value, learning_coefficient)   # runs change_weights starting with output nodes
            else:
                break   # when we reach an input nodes, we've iterated through all the non-input nodes and can exit the loop
    print("100 Percent Complete")
    print("]", end="\n")   # ends the loading bar
```

This function applies all our functions of learning, and does the bulk of the work in our program. The first function of learning it calls is “calculate_net” which is a method of the ‘Node’ class:

```
def calculate_net(self, data_input=0):
    self.net = 0
    if not self.is_input:  # Checks if the node is an input node, input nodes don't have weights, if it's not an input node the program continues.
        self.net = (sum([a * b for a, b in zip([x.output_value for x in self.inputs], self.weights[1:])]))
        self.net += self.weights[0]  # Finally the offset value is added
    else:
        self.net = data_input  # If the node is an input node, the net value is simply the input
    self.compute_output()
```

The list comprehension on the fourth line is quite complex, but it's basic function is to simply multiply the output value of any inputs to a node by that node's weights.

The function zip takes two iterables (lists in this case) and forms an iterable of tuples that are combinations of those two iterables:

```
>>> list1 = [1,2,3,4]
>>> list2 = [5,6,7,8]
>>> list(zip(list1, list2))
[(1, 5), (2, 6), (3, 7), (4, 8)]
>>>
```

The reason to use zip, instead of a FOR loop is its speed. Python's inbuilt functions are written in c, a compiled language, which makes them much faster than any alternative you would write in Python. This is because Python's assembler need not convert the function to machine code, as it's already been compiled. Throughout the code I've tried to use as many inbuilt functions as possible, in an attempt to decrease runtime.

```
[a * b for a, b in zip([x.output_value for x in self.inputs], self.weights[1:])]
```

This list comprehension unpacks all the tuples, and multiplies their values together:

```
>>> list1 = [1,2,3,4]
>>> list2 = [5,6,7,8]
>>> [a*b for a,b in zip(list1, list2)]
[5, 12, 21, 32]
>>>
```

And then sum() returns the sum of all the values in the list (another inbuilt function written in compiled C code.):

```
>>> list1 = [1,2,3,4]
>>> list2 = [5,6,7,8]
>>> sum([a*b for a,b in zip(list1, list2)])
70
>>>
```

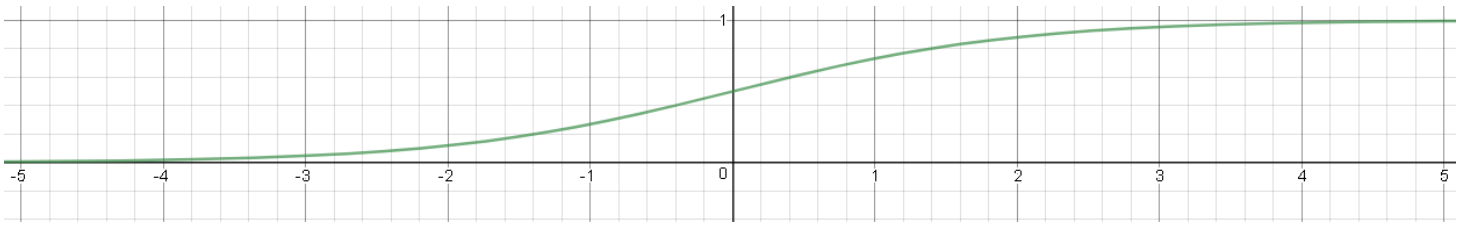
We then call the compute output function, which simply finds our output value (in this case sigmoid) and stores it:

```
def compute_output(self):
    if self.is_input:
        self.output_value = self.net
    elif self.net < -10:
        self.output_value = 0
    elif self.net > 10:
        self.output_value = 1
    else:
        self.output_value = 1/(1 + math.e**(-self.net))
```

The bounding values used here are simply to avoid having to calculate the exact value of the sigmoid when it's unnecessary. While the sigmoid is complex and changing between -2 and 2, it gets very close to 1 past 10.

	x									
	1	2	3	4	5	6	7	8	9	10
S(x)	0.731059	0.880797	0.952574	0.982014	0.993307	0.997527	0.999089	0.999665	0.999877	0.999955

As you can see, as x increases the sigmoid gets closer and closer to 1. For values larger than 10, this approximation is correct to 5 decimal places – precision greater than 5 decimal places has little to no effect on the success of learning.



The next function called in runner is change_weights, this function is where the backpropagation equations discussed earlier come into play:

```
def change_weights(self, desired_value, learning_coefficient):
    if self.is_output:
        if self.row == desired_value:   # each output neuron corresponds to one digit
            desired_value = 0.98
        else:
            desired_value = 0.02   # desired values close to 1 and 0 to improve speed convergence

        self.old_weights = self.weights[:]   # copy previous weights for later use

        self.delta_rule = self.compute_derivative_of_output() * self.calculate_derivative_of_error_function(
            desired_value)   # compute delta rule

        change_in_weights = [input_node.output_value * self.delta_rule * learning_coefficient for input_node in (self.inputs)]
                                                # find the change in weights


        self.weights = [self.weights[0]] + [sum(x) for x in zip(self.weights[1:], change_in_weights)]

        self.weights[0] += self.delta_rule * learning_coefficient

    else:
        self.old_weights = self.weights[:]

        propagation_value = sum([output_node.old_weights[self.row + 1] * output_node.delta_rule for output_node in self.connections[1:]])
                                                                #   apply backpropagation equation
        self.delta_rule = self.compute_derivative_of_output() * propagation_value

        self.weights = [self.weights[0]] + [sum(x) for x in zip(self.weights[1:], change_in_weights)]

        self.weights[0] += self.delta_rule * learning_coefficient
```

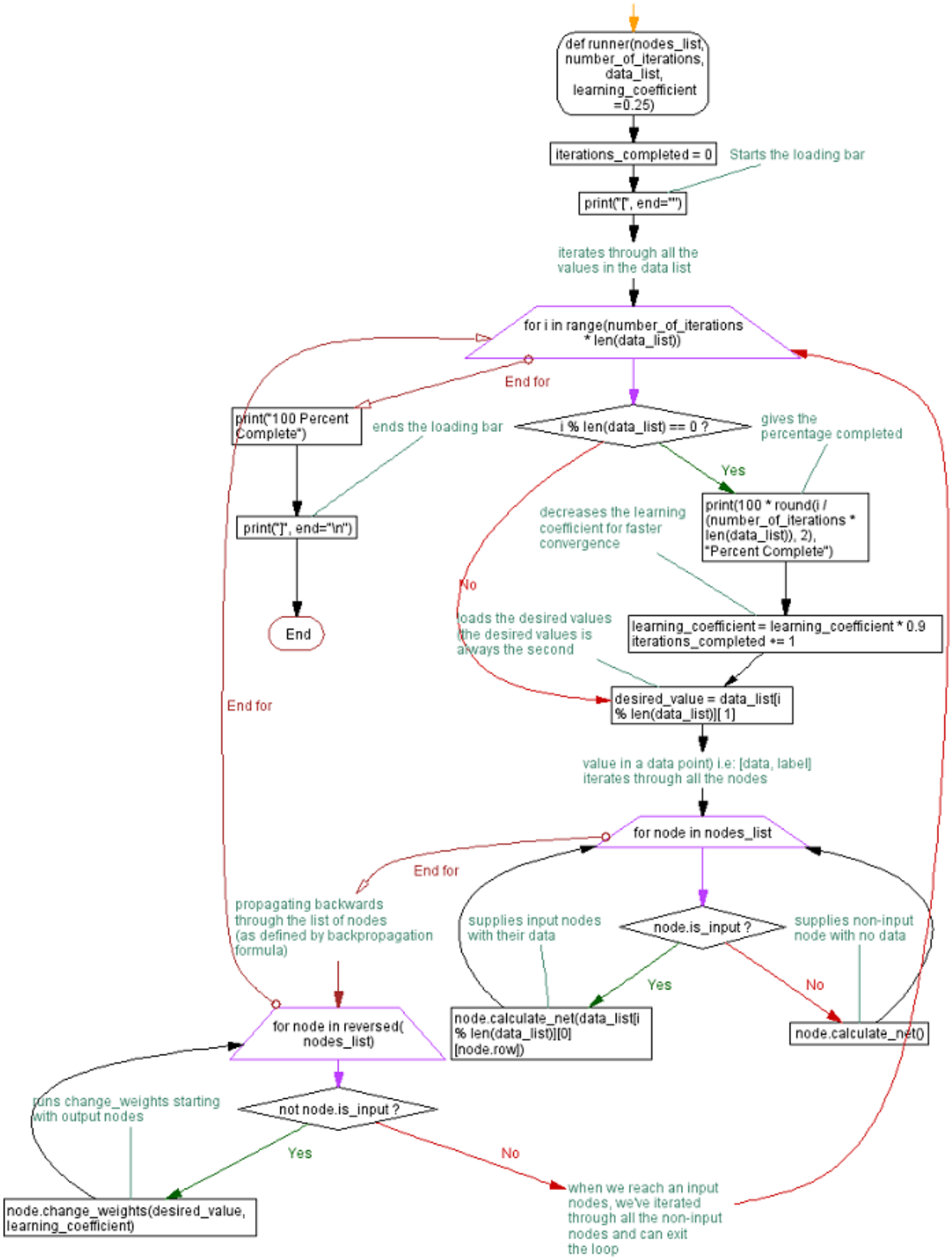


Figure 24 - Flow chart for Runner()

First of all the desired value is set to one (or close to one), if the node being evaluated is on the row corresponding to the desired value. For example if we were classifying the number 4, we would want the fourth node in the final layer to output 1 and all the other nodes to output 0.

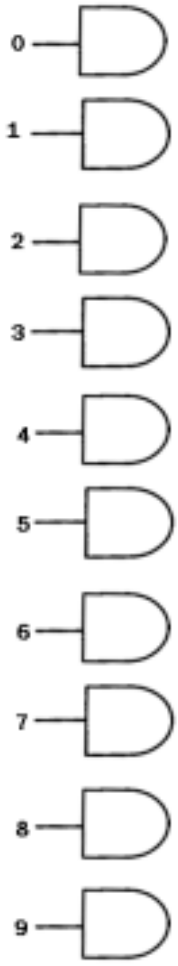


Figure 25 - Output Nodes

We then set the desired value **close** to 0 and 1 due to its improvement in convergence speed: "set the target values to be within the range of the sigmoid, rather than at its asymptotic values"[14]

Having done that we save our old weights ([:] is the fastest way to copy a list in Python).

Finding the change in weights again makes use of zip, however this time we're adding the values in the tuples rather than multiplying them:

```
>>> list1 = [1,2,3,4]
>>> list2 = [5,6,7,8]
>>> [sum(x) for x in zip(list1, list2)]
[6, 8, 10, 12]
>>>
```

All of the functions used here are consequence of the equations found earlier.

In the main code base we then run:

```
time_taken = time.time() - start  # Saves the time taken
file = open("timings.txt", "a")
file.write("\n"+str(total_iterations)+" iterations of " +str(pattern_number)+" images, completed in "+str(time_taken)+" seconds")
file.close()
```

This finds the time taken and then writes it (along with the number of images trained on and the total iterations completed) to a file. I added this so I could estimate the amount of time the model would take for larger image values.

[14] http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

Having completed our defined number of iterations, we then call run_again()

```
while run_again():

    iterations = get_no_of_iterations(iterations, nodes, data_list)
    current_start = time.time()  # Starting a second timer again after user input
    runner(nodes, iterations, data_list, default_learning_coefficient)
    total_iterations += iterations
    print("A total of " + str(total_iterations) + " iterations have been computed.")
    show(nodes, data_list)
    if print_time:   # print the time taken, print_time defined at run time
        print("Total Time taken:", time.time() - start, "Time for current iteration:", time.time() - current_start)
```

This function asks the user if they would like to continue improving the model, and if the user chooses to - runs all the aforementioned functions again.

```
def run_again():
    user_input = input("Would you like to run the code again? Press enter if you would\n" + " --> ")   # get user input
    if user_input.lower() in ["true", "yes", "sure", "i would", "okay", "es", "", "y"]:   # if the user would like to run again
        return True   # return true
    else:
        return False   # otherwise return false
```

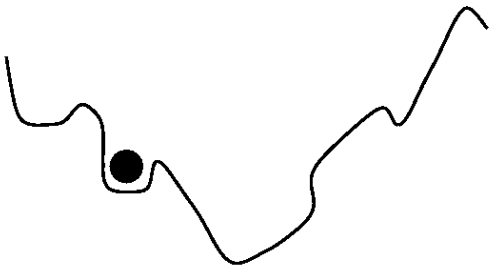
Having made these functions I implemented one other feature to improve convergence, called momentum:

38

$$\Delta w(t+1) = \eta \frac{\partial E_{t+1}}{\partial w} + \mu \Delta w(t),$$

This equation says that for our change in weights, we should calculate our error (and multiply it by our learning coefficient) as normal and then add our previous weight change (multiplied by a small constant).

Consider a ball falling down a hill, should it reach a divet in the hill – if its momentum is high enough it will be able to escape. On top of that, as it moves down the hill it picks up speed.



So by implementing momentum to train our model, we can break out of local minima and find our way to the **global** minima. That is, we want to find the lowest error possible for our model, as some solutions reached will be better than others.

To implement this I just had to change a couple of things in the function change_in_weights():

```
change_in_weights = [
    x.output_value * self.delta_rule * learning_coefficient + 0.02 * self.old_weight_change[placeholder] for
    placeholder, x in enumerate(self.inputs)]   # finds our delta rule (using momentum)

self.old_weight_change = change_in_weights[:]   # saves change in weights for use in momentum
```

And:

```
propagation_value = sum([x.old_weights[self.row + 1] * x.delta_rule for x in self.connections[1:]])
self.delta_rule = self.compute_derivative_of_output() * propagation_value

change_in_weights = [
    x.net * self.delta_rule * learning_coefficient + 0.02 * self.old_weight_change[placeholder] for
    placeholder, x in enumerate(self.inputs)]   # uses momentum
```

I also found a far better activation function, specifically defined for the MNIST dataset:

$$f\left(net_p\right) = max\left(0,\ net_p\right) + \cos\left(net_p\right)$$

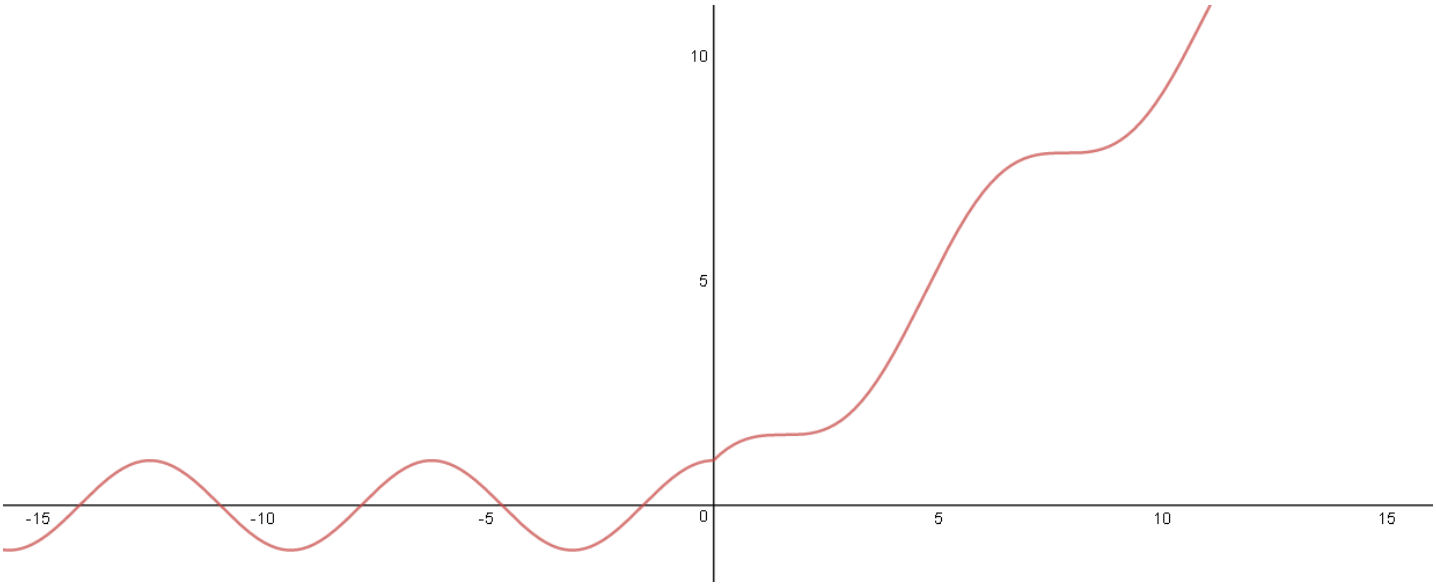
For which the derivative is:

$$f'\left(net_p\right) = \begin{cases} -\sin\left(net_p\right) & x<0 \\ 1-\sin\left(net_p\right) & x>0 \end{cases}$$

To implement this activation function I made these changes:

```
if self.is_output:
    if self.row == desired_value:   # checks if the node is the one we want to fire
        desired_value = 0.98   # if so we want a high output
    else:
        desired_value = -0.98


def compute_output(self):   # finds the value of our activation function

    if self.is_input:
        self.output_value = self.net   # our input node's output values should be their net
    else:
        self.output_value = max(0, self.net) + math.cos(self.net)

def compute_derivative_of_output(self):
    if self.net < 0:
        return -math.sin(self.net)
    else:
        return 1 - math.sin(self.net)
```

The function looks like this:



This activation function performs better than any previously discussed activation function, when classifying MNIST digits. Additionally it is faster to compute than the sigmoid or hyperbolic tangent.

Backpropagation Technical Solution

```
import math
import os  # Used to search directory when saving or loading files
import random  # Used to generate random numbers
import statistics
import time  # Used get runtime

from mnist import MNIST  # Used to unpack the MNIST database


class Node:
    def __init__(self, row, layer, is_input=False, is_output=False):

        self.connections = []  # Stores the nodes this node outputs to
        self.inputs = []  # Stores the nodes this node receives input
from
        self.row = row  # Defines the node's row
        self.layer = layer  # Defines the node's layer
        self.is_input = is_input  # Defines if the node is an input
node
        self.is_output = is_output  # Defines if the node is an output
node
        self.weights = []  # Stores the node's weights
        self.old_weights = []  # Stores the node's weights before
they're changes, for use in backpropagation.
        self.net = 0  # Stores the node's net value
        self.output_value = 0  # Stores the node's output value
        self.change_in_weight = 0  # Stores the node's change in weight
value
        self.delta_rule = 0  # Stores the node's delta rule
        self.old_weight_change = []

    def create_old_changes(self):
        if not self.is_input:  # creates a list of weight changes with
all zeroes
            self.old_weight_change = [0 for x in
                                      range(len(self.inputs))]  # so we
have a list of weight changes when the first
            # momentum calculation is done

    def add_connections(self, desired_row, desired_layer, node_list,
network_structure):
        if not (desired_row == self.row and desired_layer ==
self.layer):  # if not searching for itself
            node_id = sum(network_structure[0:desired_layer]) +
desired_row  # find the node to connect to
            desired_node = node_list[node_id]  # get the node in the
node list

            if desired_node.row == desired_row and desired_node.layer
== desired_layer:  # check that was the node we're looking for
                self.connections.append(desired_node)  # add the node
to self.connections
```

```
                desired_node.inputs.append(self)  # adds itself to the
nodes list of inputs
                return None  # exit

            print("add_connections calling a node that doesnt exist or
the wrong node")  # Error Chekcing (:
            print(desired_row, desired_layer)

    def show_connections(self):
        temp = []  # Creates a temporary list
        for i in self.connections:  # Iterates through the nodes
connections
            temp.append((i.row,
                         i.layer))  # Adds each connection to a
temporary list in an easy to read fashion (in vector form).
        return temp  # returns the temporary list

    def show_inputs(self):
        temp = []  # Creates a temporary list
        for i in self.inputs:  # Iterates through the nodes connections
            temp.append((i.row,
                         i.layer))  # Adds each connection to a
temporary list in an easy to read fashion (in vector form).
        return temp  # returns the temporary list

    def add_weights(self, weights):
        if (len(weights) - 1 == len(
                self.inputs)) and not self.is_input:  # Checking if the
number of inputs the node has equals the number of weights (+1 for the
offset weight)
            self.weights = weights  # Sets its weights to the weights
provided
            return None  # Then Exits
        print("add_weights has been supplied more weights than the "  #
Error Chekcing (:
              "node has connections or its an input node")

    def calculate_net(self, data_input=0):
        self.net = 0
        if not self.is_input:  # Checks if the node is an input node,
input nodes don't have weights, if it's not an input node the program
continues.
            self.net = (sum([a * b for a, b in zip([x.output_value for
x in self.inputs], self.weights[1:])]))
            self.net += self.weights[0]  # Finally the offset value is
added
        else:
            self.net = data_input  # If the node is an input node, the
net value is simply the input
        self.compute_output()

    def compute_output(self):  # finds the value of our activation
function
```

```
        if self.is_input:
            self.output_value = self.net  # our input node's output
values should be their net
        else:
            self.output_value = max(0, self.net) + math.cos(self.net)

    def compute_derivative_of_output(self):
        if self.net < 0:
            return -math.sin(self.net)
        else:
            return 1 - math.sin(self.net)

    def calculate_derivative_of_error_function(self, desired_value):

        return desired_value - self.output_value

    def change_weights(self, desired_value, learning_coefficient):
        if self.is_output:
            if self.row == desired_value:  # checks if the node is the
one we want to fire
                desired_value = 0.98  # if so we want a high output
            else:
                desired_value = -0.98

            self.old_weights = self.weights[:]  # copies old weights
list

            self.delta_rule = self.compute_derivative_of_output() *
self.calculate_derivative_of_error_function(
                desired_value)  # computes our delta rule

            change_in_weights = [
                x.output_value * self.delta_rule * learning_coefficient
+ 0.02 * self.old_weight_change[placeholder] for
                placeholder, x in enumerate(self.inputs)]  # finds our
delta rule (using momentum)

            self.old_weight_change = change_in_weights[:]  # saves
change in weights for use in momentum

            self.weights = [self.weights[0]] + [sum(x) for x in
zip(self.weights[1:], change_in_weights)]

            self.weights[0] += self.delta_rule * learning_coefficient

        else:
            self.old_weights = self.weights[:]

            propagation_value = sum([x.old_weights[self.row + 1] *
x.delta_rule for x in self.connections[1:]])
            self.delta_rule = self.compute_derivative_of_output() *
propagation_value

            change_in_weights = [
```

```
                x.net * self.delta_rule * learning_coefficient + 0.02 *
self.old_weight_change[placeholder] for
                placeholder, x in enumerate(self.inputs)]  # uses
momentum
            self.weights = [self.weights[0]] + [sum(x) for x in
zip(self.weights[1:], change_in_weights)]

            self.weights[0] += self.delta_rule * learning_coefficient


def network_creator(node_list, structure_list):
    for layer, number_of_nodes in enumerate(structure_list):  #
iterates through our list of nodes
        if layer == 0:  # if an input node
            for row in range(number_of_nodes):
                node_list.append(Node(row, layer, is_input=True))  # we
want to initialise with is_input set to True
        elif layer == len(structure_list) - 1:  # if an output node
            for row in range(number_of_nodes):
                node_list.append(Node(row, layer, is_input=False,
                                      is_output=True))  # we want to
initialise with is_output set to True
        else:
            for row in range(number_of_nodes):
                node_list.append(Node(row, layer))  # otherwise
initialize as normal


def connect_all_nodes(nodes, network__structure):
    for layer_no, layer in enumerate(network__structure[:-1]):  #
iterates through our network structure
        for output_node in nodes[0:layer]:  # iterates through all the
nodes in the layer

            node_up_to = sum(network__structure[0:(layer_no + 1)])  #
gets the number of nodes before the layer

            for input_node in nodes[node_up_to:node_up_to +
network__structure[layer_no + 1]]:
                output_node.add_connections(input_node.row,
input_node.layer, nodes,
                                            network__structure)  #
connects all the nodes


def generate_weights(nodes, reset=False):
    for node in nodes:
        if node.layer != 0:  # if not an input
            for inputs in range(len(node.inputs) + 1):
                if reset:
                    node.weights[inputs] = random.gauss(0, len(
                        node.inputs) ** -0.5)  # generates weights as
according to Efficient BackProp, to improve convergence speed
                else:
```

```
                    node.weights.append(random.gauss(0,
len(node.inputs) ** -0.5))
            node.weights[0] = 0


def write_weights_to_file(nodes, data_list, network_structure, name):
    file = open(name + ".txt", "w")  # opens the specified file
    file.write("NO_DATA\n")  # writes our data list to the file
    for node_layer in range(nodes[len(nodes) - 1].layer):  # gets the
number of nodes and iterates that many times
        line = ""
        for x in nodes:
            if x.layer == (node_layer + 1):  # finds nodes of specific
layers (dont want input nodes)
                line += str(x.weights) + ":"  # writes adds their
weights to a string
        line = line[:-1]  # removes the final : from the string
        file.write(line + "\n")  # writes the string to the file, and
adds a new line
    file.write(str(network_structure)[1:-1])


def save_user_defined_file(nodes, data_list, network_structure):
    user_input = input("Would you like to save this model? Y or N \n --
> ")
    if user_input.lower() in ["y", "true", "yes", "sure", "i would",
"okay", "es"]:
        name = input("Please enter the desired name of your file.
(Without .txt)\n --> ")
        while (name + ".txt") in os.listdir():  # searches the
directory for the user defined file name
            user_input = input(
                "Sorry theres another file with that name saved
already. Would you like to try another? Y or N\n --> ")
            if user_input.lower() in ["y", "true", "yes", "sure", "i
would", "okay", "es"]:
                name = input("Please enter the desired name of your
file. (Without .txt)\n --> ")
            else:
                break
        write_weights_to_file(nodes, data_list, network_structure,
name)


def run_again():
    user_input = input("Would you like to run the code again? Press
enter if you would\n" + " --> ")  # get user input
    if user_input.lower() in ["true", "yes", "sure", "i would", "okay",
"es", "",
                              "y"]:  # if the user would like to run
again
        return True  # return true
    else:
        return False  # otherwise return false
```

```
def get_no_of_iterations(iterations, nodes_list, data_list):
    print("Default Iterations = " + str(iterations))
    user_input = input("How many Iterations would you like, press enter
for default.\n --> ").lower()
    while True:
        try:  # try to turn user in put to an integer
            user_input = int(user_input)
            if user_input > 0:
                print("Running " + str(user_input) + " iterations")
                return user_input
            elif user_input == 0:
                show(nodes_list, data_list)  # if the user selected
zero iterations, show output from the current model
                return 0
            else:
                print("Input must be greater than zero!")
                user_input = input("How many Iterations would you like,
press enter for default.\n --> ")
        except:
            print("Running " + str(
                iterations) + " iterations")  # if the suer gave bad
input, run default number of iterations
            return iterations


def set_weights(nodes, weight_data):
    for node in nodes:
        if node.layer != 0:  # if the node isn't an input node
            node.weights = weight_data[node.layer - 1][node.row]  # set
the weights to the data supplied


def get_file_data(input_file_name):
    file = open(input_file_name + ".txt", "r")  # open the user defined
file
    file_line_list = file.read().split("\n")  # read the file
    layers = []
    network_structure = [int(file_line_list[len(file_line_list)-1][0])]
# stores the number of input nodes

    for line in file_line_list[1:-1]:  # removes the brackets at the
end of every list
        layers.append(line.split(":"))

    for layer in layers:
        network_structure.append(len(layer))
        for node_no, node in enumerate(layer):
            node = layer[node_no] = eval(node)
            for weight_number, weight in enumerate(node):
                node[weight_number] = float(weight)

    return network_structure, layers
```

```
def runner(nodes_list, number_of_iterations, data_list,
learning_coefficient=0.25):
    iterations_completed = 0
    print("[", end="")  # Starts the loading bar
    for i in range(number_of_iterations * len(data_list)):  # iterates
through all the values in the data list
        if i % len(data_list) == 0:
            print(round(100 * i / (number_of_iterations *
len(data_list)), 2),
                  "Percent Complete")  # gives the percentage completed
            learning_coefficient = learning_coefficient * 0.9 #
decreases the learning coefficient for faster convergence
            iterations_completed += 1
        desired_value = data_list[i % len(data_list)][
            1]  # loads the desired values (the desired values is
always the second
        # value in a data point) i.e: [data, label]
        for node in nodes_list:  # iterates through all the nodes
            if node.is_input:
                node.calculate_net(data_list[i %
len(data_list)][0][node.row])  # supplies input nodes with their data
            else:
                node.calculate_net()  # supplies non-input node with no
data

        for node in reversed(
                nodes_list):  # propagating backwards through the list
of nodes (as defined by backpropagation formula)
            if not node.is_input:
                node.change_weights(desired_value,
                                    learning_coefficient)  # runs
change_weights starting with output nodes
            else:
                break  # when we reach an input nodes, we've iterated
through all the non-input nodes and can exit the loop
    print("100 Percent Complete")
    print("]", end="\n")  # ends the loading bar


def show(nodes_list, data_list):
    final_layer = nodes_list[len(nodes_list) - 1].layer  # gets the
final layer of the neural network
    good_guesses = 0
    for placeholder in range(len(data_list)):  # iterates through all
the values in the list
        desired_value = data_list[placeholder % len(data_list)][1]  #
gets the desired output for the values in the list
        for node in nodes_list:  # propagates forward to calculate
error at output
            if node.is_input:
                node.calculate_net(
                    data_list[placeholder %
```

```
len(data_list)][0][node.row])  # gives the input nodes their input
            else:
                node.calculate_net()

        for node in nodes_list[-10:]:  # iterates through output nodes
            last_layer_outputs = [node.output_value for node in
                                  nodes_list[-10:]]  # gets the outputs
of all the output nodes
            if node.layer == final_layer:  # if the node is an output
node
                print(node.output_value, desired_value)  # show the
user it's output
                if node.row == desired_value:
                    correct_node_output = node.output_value  # show the
output of the node we want to be firing
                    if node.output_value == max(last_layer_outputs):
                        good_guesses += 1  # if the right node fired,
add to good_guesses

        highest_output_index = last_layer_outputs.index(
            max(last_layer_outputs))  # the index of the node with the
highest output
        print("The correct node ouputted", correct_node_output, "and
the model predicted", highest_output_index,
              end=" ")
        if highest_output_index == desired_value:
            print("which was the correct value", end="\n")
        else:
            print("which was the incorrect value", end="\n")

    print("We had", good_guesses, "good guesses or a " +
str(good_guesses / (len(data_list)) * 100) + "% success rate")
    if good_guesses / (len(data_list)) > 0.8:
        return True


def unpack_mnist(number_of_images=100):
    mndata = MNIST('C:\\Users\\magee\\Desktop\\NEA-20190405T164125Z-
001\\Method\\MNIST')

    images, labels = mndata.load_training()

    data_list = []

    ##        for i in range(number_of_images):
    ##            print(MNIST.display(images[i]))

    for i in range(number_of_images):
        mean = statistics.mean(images[i])
        std_dev = statistics.stdev(images[i])
        for placeholder, data_point in enumerate(images[i]):
            images[i][placeholder] = (data_point - mean) / std_dev

    for i in range(number_of_images):
```

```
        data_list.append([images[i], labels[i]])

    return data_list


def get_user_input(nodes):
    file_name = input("Please enter the desired name of your file.
(Without .txt)\n --> ")

    while (file_name + ".txt") not in os.listdir():  # searches the
directory for the file the user entered
        #   if the file isn't found, ask the user for input again
        user_input = input("Sorry that file isn't in this directory.
Would you like to try another file? Y or N\n --> ")
        if user_input.lower() in ["y", "true", "yes", "sure", "i
would", "okay", "es"]:  # if they would
            file_name = input("Please enter the desired name of your
file. (Without .txt)\n --> ")  # try again
        else:
            return False  # otherwise return false

    try:  # catch any errors
        data_file = open(file_name + ".txt", "r")  # open their file
        file_data = data_file.read().split(",")  # split the file
according to commas
        # this is how data from the drawing tool is formatted
        file_data = list(map(float, file_data))  # converts all the
data to floats
        mean = statistics.mean(file_data)  # gets the mean of the input
data
        std_dev = statistics.stdev(file_data)  # gets the standard
deviation of the input data
        for placeholder, data_point in enumerate(file_data):
            file_data[placeholder] = (data_point - mean) / std_dev  #
normalizes the data
    except:
        print("That file type isn't supported, data must be seperated
by commas")
        return True

    show_output_from_input(nodes, file_data)


def show_output_from_input(nodes_list, data_input):
    guess = 0
    for i in range(len(data_input)):
        for node in nodes_list:
            if node.is_input:
                node.calculate_net(data_input[node.row])
            else:
                node.calculate_net()
    largest_output = max([x.output_value for x in nodes_list[-10:]])
    for node in nodes_list:
        if node.layer == nodes_list[len(nodes_list) - 1].layer:
```

```
            print("Output value:", node.output_value)
            if node.output_value == largest_output:
                guess = node.row
    print("My guess is", guess)


def initialise(network__structure=None, data_list=None,
print_time=True, get_network=True):
    if network__structure is None:  # If no network structure is given
        network__structure = [768, 400, 400, 10]  # load a default
    if data_list is None:  # If no data is given
        pattern_number = 1
        print("Loading",pattern_number, "images")
        data_list = unpack_mnist(pattern_number)  # Load the MNIST
database

    print(network__structure)
    nodes = []
    network_creator(nodes, network__structure)  # Creats instances of
node class
    connect_all_nodes(nodes, network__structure)  # Connects all node
layers

    # !!!
    for node in nodes:
        node.create_old_changes()  # creates a list of old weight
changes for use in momentum

    if get_network:  # If the user wants to load a already created
network (variable defined at runtime)
        while True:
            try:
                network__structure, layers = get_file_data(input(
                    "Please enter the name of the file to be used as
input (Without .txt )\n --> "))  # Try and find it
                set_weights(nodes, layers)  # Try and initialize the
weights
                break
            except:
                user_input = input(
                    "An error has occured, would you like to reenter
the name Y or N \n--> ")  # Else load default data
                if user_input.lower() in ["y", "true", "yes", "sure",
"i would", "okay", "es"]:
                    continue
                else:
                    generate_weights(nodes)
                    break

    default_learning_coefficient = 0.002  # Default learning
coefficient, changes speed of learning 0.0033 is a good one
    total_iterations = 0  # Keeps track of the total iterations
computed
    # !!!
```

```
    iterations = 10  # Default no of iterations
    iterations = get_no_of_iterations(iterations, nodes, data_list)  #
Get user input

    start = time.time()  # Start the timer (After user input)
    runner(nodes, iterations, data_list, default_learning_coefficient)
# Initialises learning
    total_iterations += iterations  # Adds to the total iterations
computed

    time_taken = time.time() - start  # Saves the time taken
    file = open("timings.txt", "a")
    file.write("\n" + str(total_iterations) + " iterations of " +
str(pattern_number) + " images, completed in " + str(
        time_taken) + " seconds")
    file.close()

    show(nodes, data_list)  # Show output values for specific inputs

    if print_time:
        print("Time taken:", time_taken)  # Print the time taken

    while input("Would you like to test a value on this model? Y or N\n
--> ").lower() in ["y", "true", "yes", "sure",

"i would", "okay", "es"]:
        if not get_user_input(nodes):  # if the get_user_input function
returns false
            break  # exit the loop

    while run_again():

        iterations = get_no_of_iterations(iterations, nodes, data_list)
        current_start = time.time()  # Starting a second timer again
after user input
        runner(nodes, iterations, data_list,
default_learning_coefficient)
        total_iterations += iterations
        print("A total of " + str(total_iterations) + " iterations have
been computed.")
        show(nodes, data_list)
        if print_time:  # print the time taken, print_time defined at
run time
            print("Total Time taken:", time.time() - start, "Time for
current iteration:", time.time() - current_start)
        while input("Would you like to test a value on this model? Y or
N\n --> ").lower() in ["y", "true", "yes", "sure",

"i would", "okay", "es"]:
            if not get_user_input(nodes):  # if the get_user_input
function returns false
                break  # exit the loop
```

```
    write_weights_to_file(nodes, data_list,
network__structure,"weights")  # writes an autosave of the model to a
file
    file = open("weights", "w")
    file.close()
    save_user_defined_file(nodes, data_list, network__structure)
    print("Thank You!")


if __name__ == "__main__":
    initialise()
    # cProfile.run('initialise()', sort='cumulative')
```

Visualizer Technical Solution:

```
import tkinter as tk
from random import randint

def __init__():

    layers = []
    file_length = sum(1 for x in open("weights.txt", "r"))    # get the file
length
    file = open("weights.txt", "r")    # open the file containing all the
data of our neural network

    for line_number, line in enumerate(file):
        if line_number != 0 and line_number != file_length-1:    # if we
aren't on the first or last line
            print(line)
            print(line.rstrip().split(":"))
            layers.append(line.rstrip().split(":"))    # rstrip removes any
empty space in the string (if the user has entered some)
                                                       # and then splits
string of weights into individual lists
        elif line_number == file_length-1:    # if we are on the last line
            arrangement = [int(value) for value in line.split(",")]    # as
the network structure is written to the last line of the file
                                                                       #
turn all the values in the network stru

    layers.insert(0, [" INPUT " for x in range(arrangement[0])])    #
```

```
inserts empty strings to be converted to input

    for layer in layers:
        for placeholder, weights in enumerate(layer):
            layer[placeholder] = weights[1:-1].split(",")
            if layer[placeholder] == ["INPUT"]:
                continue
            else:
                layer[placeholder] = [float(x) for x in layer[placeholder]]
# converts all our weights from strings to floats


    root = tk.Tk()
    width = len(layers)*2*90   # gets width of the window

    top = max(map(len, layers))   # gets the largest row

    height = top*70+70   # gets height of window

    root.geometry(str(width)+"x"+str(height)+"+200+200")   # sets window
height and position

    button_window = tk.Toplevel(root)   # creates secondary window for
buttons
    button_window.geometry("100x50+78+200")

    button_window.resizable(False,False)   # makes the button window non-
resizable in the x or y plane


    y_scrollbar = tk.Scrollbar(root, orient="vertical")
    y_scrollbar.pack(side="right", fill="y")   # adds the scrollbar to the
window

    x_scrollbar = tk.Scrollbar(root, orient="horizontal")
    x_scrollbar.pack(side="bottom", fill="x")

    canvas = tk.Canvas(root, width = width, height =height,
scrollregion=(0,0,width,height))   # canvas with a scroll region the width
& height

# defined earlier

    canvas.pack()

    canvas.config(yscrollcommand=y_scrollbar.set)   # configuring the
scrollbars to scroll
    y_scrollbar.config(command=canvas.yview)

    canvas.config(xscrollcommand=x_scrollbar.set)
    x_scrollbar.config(command=canvas.xview)


    class Nodes:

        def __init__(self, row, node_layer, weights=[], is_input=False,
```

```
is_output=False):
            self.node_layer = node_layer
            self.row   = row
            self.weights = weights
            self.is_input = is_input
            self.is_output = is_output
            node_layers[self.node_layer].append(self)
            self.node_object = None
            self.centre_x = 0
            self.centre_y = 0
            self.lines = []
            self.text = None


        def draw_node(self):
            radius = 25
            self.centre_y = 30 + 2*30*self.row +30*(max(arrangement)-
len(node_layers[self.node_layer]))   # sets its centre to be in line with
other nodes
            self.centre_x = 30 + 2*100*self.node_layer
            colour = "lightgrey"
            if self.is_input:
                colour = "lightblue"
            elif self.is_output:
                colour = "orange"
            self.node_object = canvas.create_oval(self.centre_x-radius,
self.centre_y+radius, self.centre_x+radius, self.centre_y-radius,
fill=colour)
            if not self.is_input:
                text = str(round(sum(self.weights)/len(self.weights), 5))
# gets the average weight value
            else:
                text = "INPUT"
            self.text = canvas.create_text(self.centre_x, self.centre_y,
text = text, width = 60)   # writes the text on the node


        def hide_node(self):
            canvas.delete(self.node_object)
            canvas.delete(self.text)


        def connect_to_next_layer(self):   # draws lines between each node
on each layer
            for node in node_layers[self.node_layer + 1]:
                self.lines.append(canvas.create_line(self.centre_x,
self.centre_y, node.centre_x, node.centre_y))
                canvas.tag_lower(self.lines[len(self.lines) - 1])   # moves
the line behind the circle


        def hide_connections(self):
            for connection in self.lines:
                canvas.delete(connection)
```

```
    node_layers = [[] for x in range(len(layers))]

    for layer_no, layer in enumerate(layers):
        for row_no, node in enumerate(layer):
            if layer_no == 0:
                Nodes(row_no, layer_no, node, is_input=True)    #
initializes all our nodes
            elif layer_no == len(layers)-1:
                Nodes(row_no, layer_no, node, is_output=True)
            else:
                Nodes(row_no, layer_no, node)

    def show_nodes():
        show_nodes_button.config(text = "Hide Nodes")
        show_nodes_button.config(command = hide_nodes)
        for layer in node_layers:
            for node in layer:
                node.draw_node()

    def hide_nodes():
        show_nodes_button.config(text = "Show Nodes")
        show_nodes_button.config(command = show_nodes)
        for layer in node_layers:
            for node in layer:
                node.hide_node()

    def connect_nodes():
        connect_nodes_button.config(text = "Hide Connections")
        connect_nodes_button.config(command = disconnect_nodes)
        for layer in node_layers:
            for node in layer:
                if not node.is_output:
                    node.connect_to_next_layer()

    def disconnect_nodes():
        connect_nodes_button.config(text = "Show Connections")
        connect_nodes_button.config(command = connect_nodes)
        for layer in node_layers:
            for node in layer:
                node.hide_connections()



    show_nodes_button = tk.Button(button_window, text = "Hide nodes",
command = hide_nodes)
    connect_nodes_button = tk.Button(button_window, text = "Hide
Connections", command = disconnect_nodes)
    show_nodes()
    connect_nodes()
    connect_nodes_button.pack()
    show_nodes_button.pack()
    root.mainloop()
if __name__ == "__main__":
    __init__()
```

A video of the program being used:

https://youtu.be/bok1rvKIc5w

Initially I show some functionality of the drawing tool (0:00 – 0:33 seconds) – the left clicking, right clicking and clear functionality are all tested and shown to work. Then finally I draw the digit ‘1’ and save and exit the tool.

I then load a previously trained model that had been saved in a file “837outof1000.txt”, the name signifies that the model had correctly classified 837 out of 1000 images it’s training phase. This is a model with an input layer of 768 neurons, two hidden layers of 400 neurons each and finally an output layer of 10 neurons. I then output the result of 0 iterations, which is incorrect classification due to non-MNIST data being loaded.

I then load the “drawing.txt”, this is where our previously drawn 1 is stored. The model loads the data and correctly classifies it as a one!

I then show an example model visualized, and some features of the visualizer.

The "canvas cleared" pop-up was due to repeated presses of the 'c' button, not any problems with the program.

Testing:

Test Number	Being Tested:	Description	Data Type	Expected Result	Pass/Fail	Cross Reference
1	Model Trainer	Testing a previously created model as input	Typical	Load model	Pass	Test 1 Screenshot
2	Model Trainer	Testing a non-existent file as input	Erroneous	Ask again for input	Pass	Test 2 Screenshot
3	Model Trainer	Entering a negative number for number of iterations	Erroneous	Ask again for input	Pass	Test 3 Screenshot
4	Model Trainer	Entering a string for number of iterations	Erroneous	Continue with default no of iterations	Pass	Test 4 Screenshot
5	Model Trainer	Entering 10 for number of iterations	Typical	Compute 10 iterations	Pass	Test 5 Screenshot
6	Model Trainer	Entering 0 for number of iterations	Typical	Show output from untrained model	Pass	Test 6 Screenshot
7	Model Trainer	Training for 10 iterations with 100 mnist pictures	Typical	Show accurate output	Pass	Test 7 Screenshot
8	Model Trainer+Drawing Tool	Drawing a 6 and then testing it on a model trained with 100 images	Typical	Predict the incorrect number	Pass	Test 8 Screenshot
9	Model Trainer+Drawing Tool	Drawing a 6 and then testing it on a model trained with 1000 images	Typical	Predict the correct number	Pass	Test 9 Screenshot
10	Model Trainer+Visualizer	Training a model of network structure [768,2,10] and visualising it	Typical	Terrible model, but drawn correctly	Pass	Test 10 Screenshot
11	Drawing Tool	Testing the drawing tool with left clicks	Typical	Drawing tool draws black squares around cursor	Pass	Test 11 Screenshot

12	Drawing Tool	Testing the drawing tool with dragged left clicks	Typical	Drawing tool continually draws black squares	Pass	Test 12 Screenshot
13	Drawing Tool	Testing the drawing tool with right clicks	Typical	Drawing tool erases black squares around cursor	Pass	Test 13 Screenshot
14	Drawing Tool	Testing the drawing tool with right clicks	Typical	Drawing tool continually erases black squares	Pass	Test 14 Screenshot
15	Drawing Tool	Testing the clear button	Typical	Drawing tool clears the canvas	Pass	Test 15 Screenshot
16	Drawing Tool	Testing the left click on already coloured squares	Erroneous	Nothing will happen	Pass	Test 16 Screenshot
17	Drawing Tool	Testing the right click on blank squares	Erroneous	Nothing will happen	Pass	Test 17 Screenshot
18	Visualizer	Visualising a model with structure [2,2,1]	Typical	Model visualised correctly	Pass	Test 18 Screenshot
19	Visualizer	Testing the hide nodes button on the visualized model	Typical	Nodes to be hidden	Pass	Test 19 Screenshot
20	Visualizer	Testing the hide connections button on the visualized model	Typical	Connections between nodes to be hidden	Pass	Test 20 Screenshot
21	Visualizer	Testing hide connections when nodes are hidden	Erroneous	The user will be left with a blank screen to look at	Pass	Test 21 Screenshot
22	Visualizer	Testing hide nodes when connections are hidden	Erroneous	The user will be left with a blank screen to look at	Pass	Test 22 Screenshot
23	Visualizer	Testing Visualization of a [2,3,3,2] network	Typical	The model will be visualized correctly	Pass	Test 23 Screenshot
24	Model Trainer	Training Data on 1000 MNIST images with a model with structure [768, 400, 400, 10]	Typical	The model will have a high accuracy in predicting the data	Pass	Test 24 Screenshot
25	Linear Model	Linear Model Tested on page 45	Typical	Model Behaves as expected	Pass	Page 45

Testing Screenshots:

Test 1 Screenshot:

```
[784, 300, 10]
Please enter the name of the file to be used as input (Without .txt )
 --> weights
Default Iterations = 10
How many Iterations would you like, press enter for default.
 --> 0
-0.9007289293128484 5
-0.9127063257930729 5
-0.92603839550726 5
-0.9303357211610593 5
-0.940964686830357 5
0.9885429656791956 5
-0.9167149850572398 5
-0.9304151270893972 5
-0.9409118106148416 5
-0.9071323594527448 5
The correct node ouputted 0.9885429656791956 and the model predicted 5 which was the correct value
We had 1 good guesses or a 100.0% success rate
How many Iterations would you like, press enter for default.
 -->
```

Loading the model “weights.txt” and then showing its output. Its output shows training had been previously completed.

Test 2 Screenshot:

```
[784, 300, 10]
Please enter the name of the file to be used as input (Without .txt )
 --> fakefile
An error has occured, would you like to reenter the name Y or N
-->
```

User is prompted to re-enter the file name, assuming they’ve made an error in spelling.

Test 3 Screenshot:

```
[784, 300, 10]
Please enter the name of the file to be used as input (Without .txt )
 -->
An error has occured, would you like to reenter the name Y or N
-->
Default Iterations = 10
How many Iterations would you like, press enter for default.
 --> -1
Input must be greater than zero!
How many Iterations would you like, press enter for default.
 --> -1
```

The program catches that the input is erroneous and asks to enter a different value.

Test 4 Screenshot:

```
[784, 300, 10]
Please enter the name of the file to be used as input (Without .txt )
 -->
An error has occured, would you like to reenter the name Y or N
-->
Default Iterations = 10
How many Iterations would you like, press enter for default.
 --> erroneus
Running 10 iterations
[0.0 Percent Complete
10.0 Percent Complete
20.0 Percent Complete
30.0 Percent Complete
40.0 Percent Complete
50.0 Percent Complete
60.0 Percent Complete
70.0 Percent Complete
80.0 Percent Complete
90.0 Percent Complete
100 Percent Complete
]
-0.9358935006964939 5
-0.9249690451024692 5
-0.9359463871639566 5
-0.9097260762908286 5
-0.7082798957069008 5
0.99993884780299 5
-0.9357393148282362 5
-0.9119735571785338 5
-0.9240036029052576 5
-0.9351493380372499 5
The correct node ouputted 0.99993884780299 and the model predicted 5 which was the correct value
We had 1 good guesses or a 100.0% success rate
Time taken: 2.6077308654785156
Would you like to test a value on this model? Y or N
 -->
```

Incorrect data caught and default number of iterations computed.

Test 5 Screenshot:

```
[784, 300, 10]
Please enter the name of the file to be used as input (Without .txt )
 -->
An error has occured, would you like to reenter the name Y or N
-->
Default Iterations = 10
How many Iterations would you like, press enter for default.
 --> 10
Running 10 iterations
[0.0 Percent Complete
10.0 Percent Complete
20.0 Percent Complete
30.0 Percent Complete
40.0 Percent Complete
50.0 Percent Complete
60.0 Percent Complete
70.0 Percent Complete
80.0 Percent Complete
90.0 Percent Complete
100 Percent Complete
]
-0.909079377871223 5
-0.9142375970982213 5
-0.9076246743773057 5
-0.9237311236971579 5
1.5696479964535266 5
0.9608059600116907 5
-0.9310449292315427 5
-0.9133647393721822 5
-0.918560238457298 5
-0.9240147602835632 5
The correct node ouputted 0.9608059600116907 and the model predicted 4 which was the incorrect value
We had 0 good guesses or a 0.0% success rate
Time taken: 2.6207375526428223
Would you like to test a value on this model? Y or N
 -->
```

10 Iterations completed (The wrong value is reached)

Test 6 Screenshots:

```
[784, 300, 10]
Please enter the name of the file to be used as input (Without .txt )
 -->
An error has occured, would you like to reenter the name Y or N
-->
Default Iterations = 10
How many Iterations would you like, press enter for default.
 --> 0
1.5615035623703326 5
1.46602356695686 5
0.7614504705779914 5
1.5101918159069558 5
0.9897469204788905 5
0.9935645748396514 5
-0.05828835524391118 5
0.8350846065177767 5
0.8478195587867745 5
0.5338844301000677 5
The correct node ouputted 0.9935645748396514 and the model predicted 0 which was the incorrect value
We had 0 good guesses or a 0.0% success rate
How many Iterations would you like, press enter for default.
 -->
```

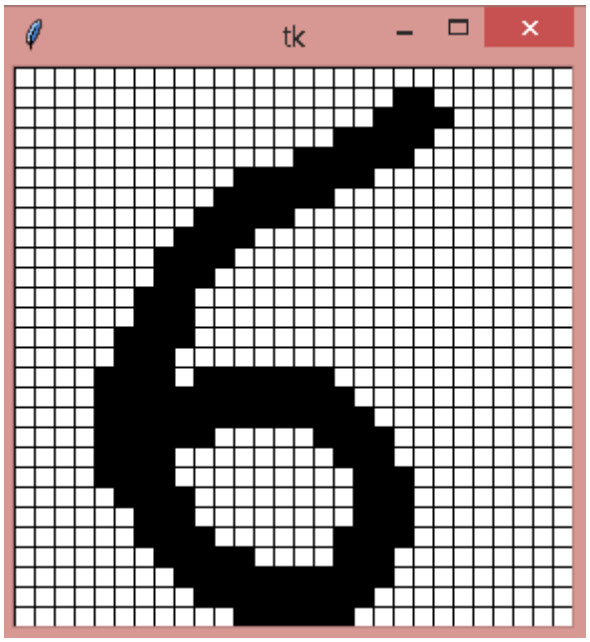
The output of the untrained model (loaded with randomly generated weights) shown.

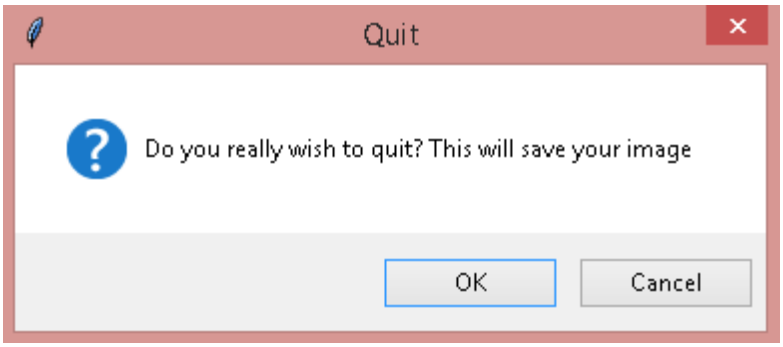
Test 7 Screenshots:

```
The correct node ouputted 0.9932367632793223 and the model predicted 6 which was
 the correct value
-0.49243747157973444 8
-0.9722894739069654 8
-0.4138482999368428 8
-0.5007233802123527 8
-0.6938856004474205 8
-0.7721844471921759 8
-0.753338178950428 8
-0.9122485506126203 8
-0.7387960261093074 8
-0.8253857078695601 8
The correct node ouputted -0.7387960261093074 and the model predicted 2 which wa
s the incorrect value
0.5149105580095599 0
-0.9998138528440881 0
-0.6914254430516497 0
-0.6582248677043762 0
-0.7001956653458864 0
-0.9295081412466738 0
-0.9204429648388204 0
-0.9868524021403502 0
-0.4754730692806174 0
-0.9419998915663723 0
The correct node ouputted 0.5149105580095599 and the model predicted 0 which was
 the correct value
-0.8754015110270791 7
-0.892588730286663 7
-0.8662530507287168 7
-0.9998245949603468 7
-0.9863740840311008 7
-0.9676608503460796 7
-0.9384239582030475 7
0.6684293127568726 7
-0.8466324965736313 7
-0.38478606184205133 7
The correct node ouputted 0.6684293127568726 and the model predicted 7 which was
 the correct value
-0.889428344451194 8
-0.9055845861881824 8
-0.9914976826151758 8
-0.9607526789458845 8
-0.8613063140958258 8
-0.8041194405776969 8
-0.9999558987862397 8
-0.9968306657850139 8
0.3806762679645224 8
-0.9434946677920226 8
The correct node ouputted 0.3806762679645224 and the model predicted 8 which was
 the correct value
-0.9836869516666068 3
-0.7803125519895416 3
-0.6973666315310135 3
0.33367060036387314 3
-0.9982126397127505 3
-0.9101060128442737 3
-0.9920219811630552 3
-0.8810238161922794 3
-0.8784298118129802 3
-0.9782681088756492 3
The correct node ouputted 0.33367060036387314 and the model predicted 3 which wa
s the correct value
-0.99700394751343 1
0.7352343510669812 1
-0.7937819533451694 1
-0.7764870854181373 1
-0.6065190606079887 1
-0.8441423209449356 1
-0.9952585471980597 1
-0.8047219232480113 1
-0.7256842759004383 1
-0.8507473827749285 1
The correct node ouputted 0.7352343510669812 and the model predicted 1 which was
 the correct value
We had 93 good guesses or a 93.0% success rate
Time taken: 64.51379084587097
Would you like to test a value on this model? Y or N
 -->
```

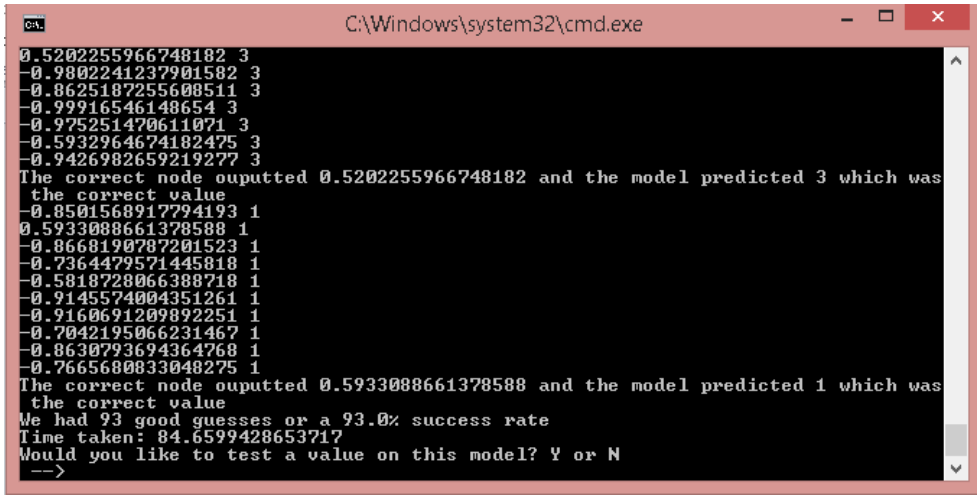
Data classified with a 93% success rate.

Test 8 Screenshots:

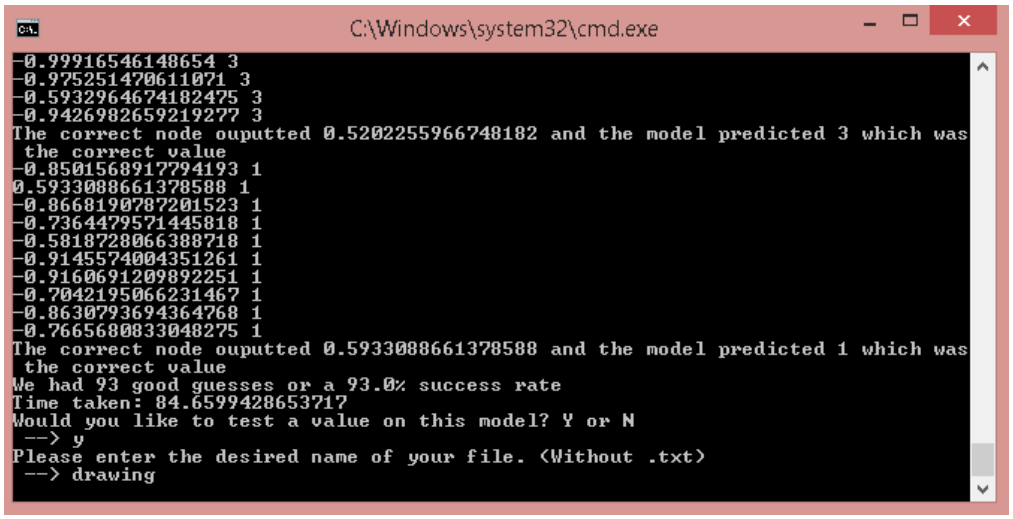




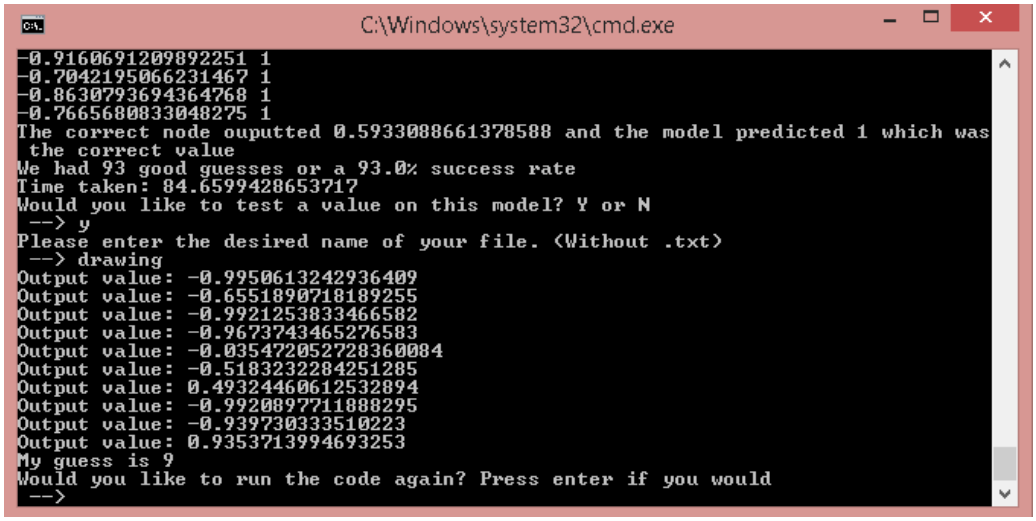
Drawing of a six created and saved.



Model trained on 100 MNIST images

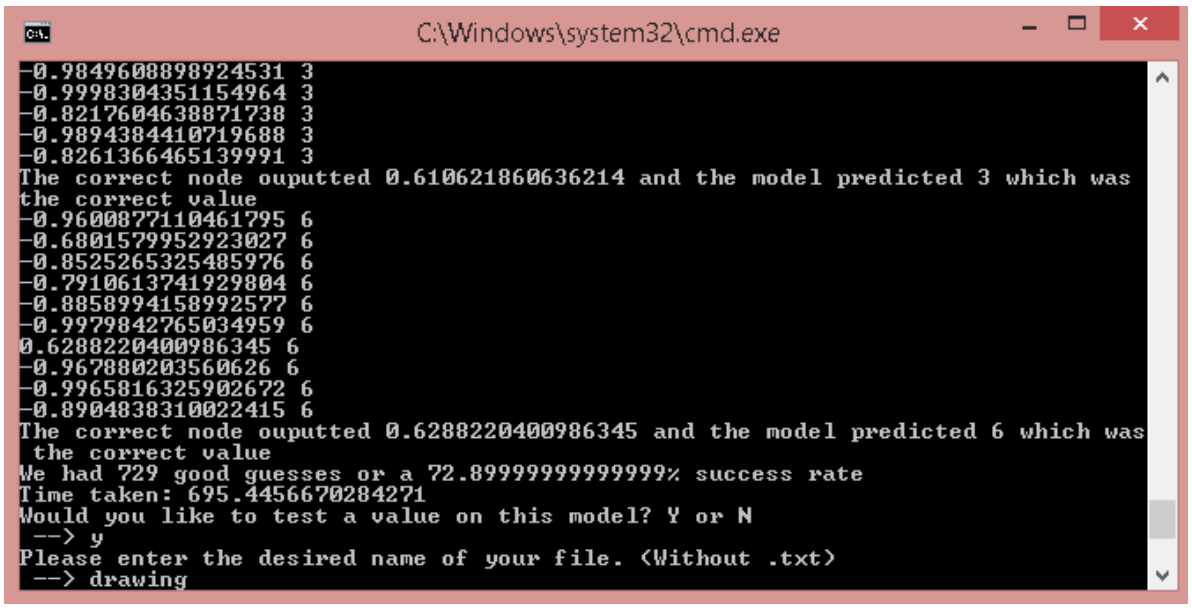


Drawing given as input

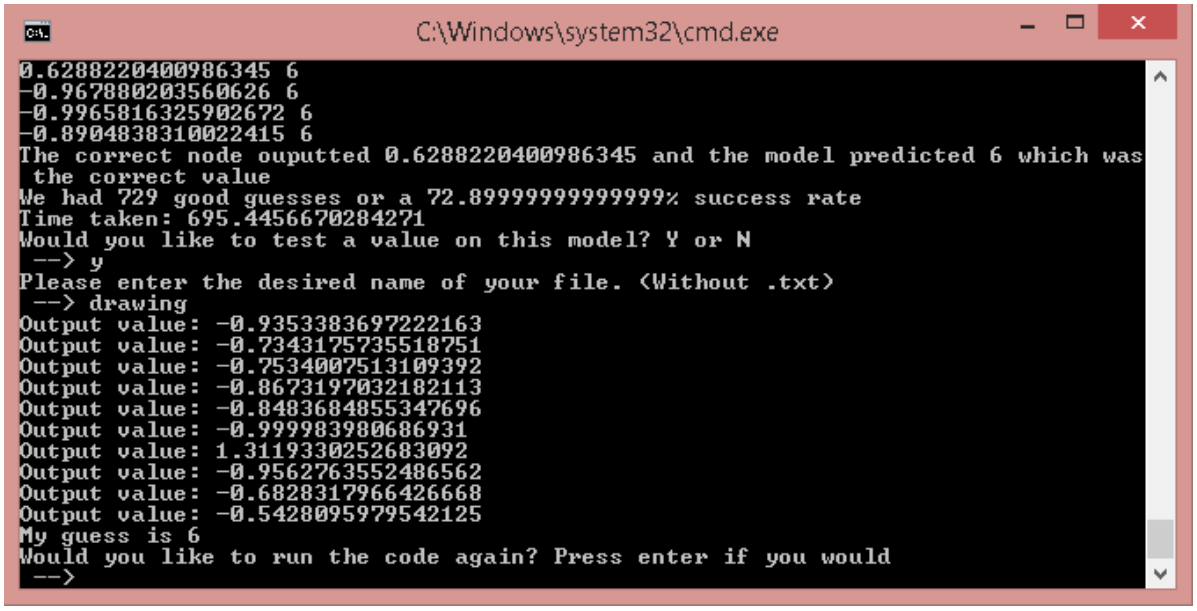


Our input was classified incorrectly (100 images is a small amount for machine learning, and the output for 6 was 0.49, which was the second highest. So for only 100 images I am impressed)

Test 9 Screenshots:

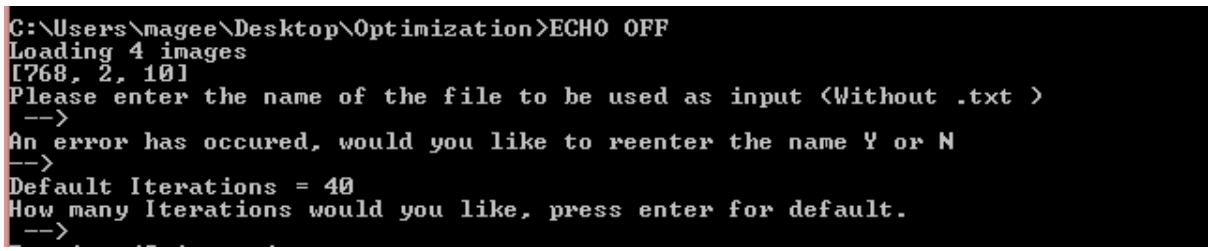


Training a model (briefly) to a 73% success rate, then giving it the previously drawn 6 as input.

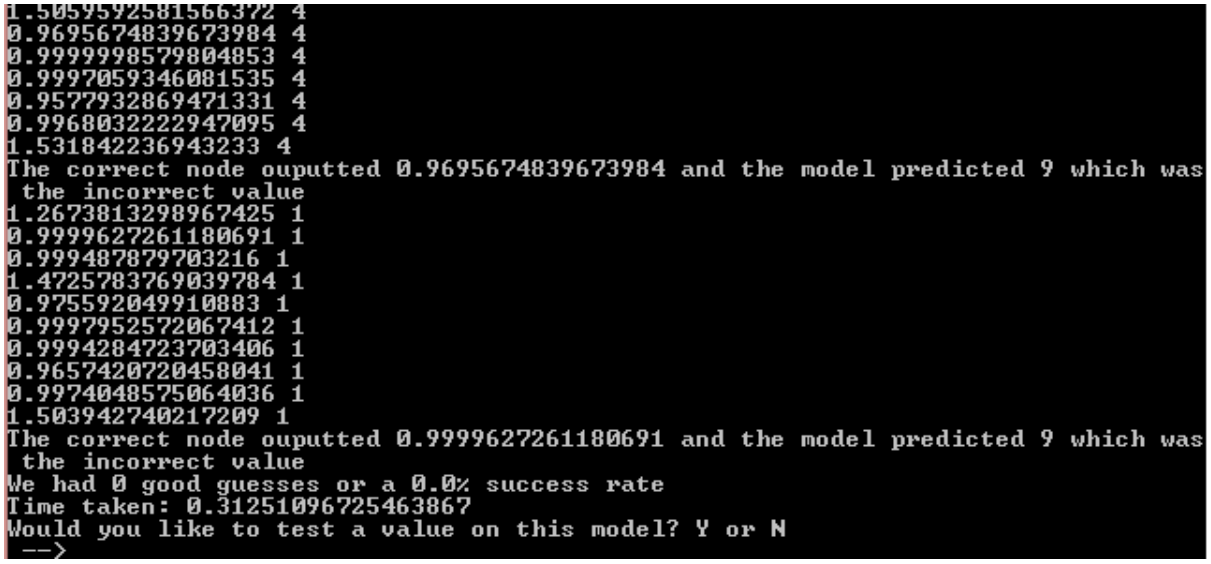


The model confidently predicted the input to be a 6 (correctly), the next closest guess was a 9 (the previous output) showing the learning process.

Test 10 Screenshots:

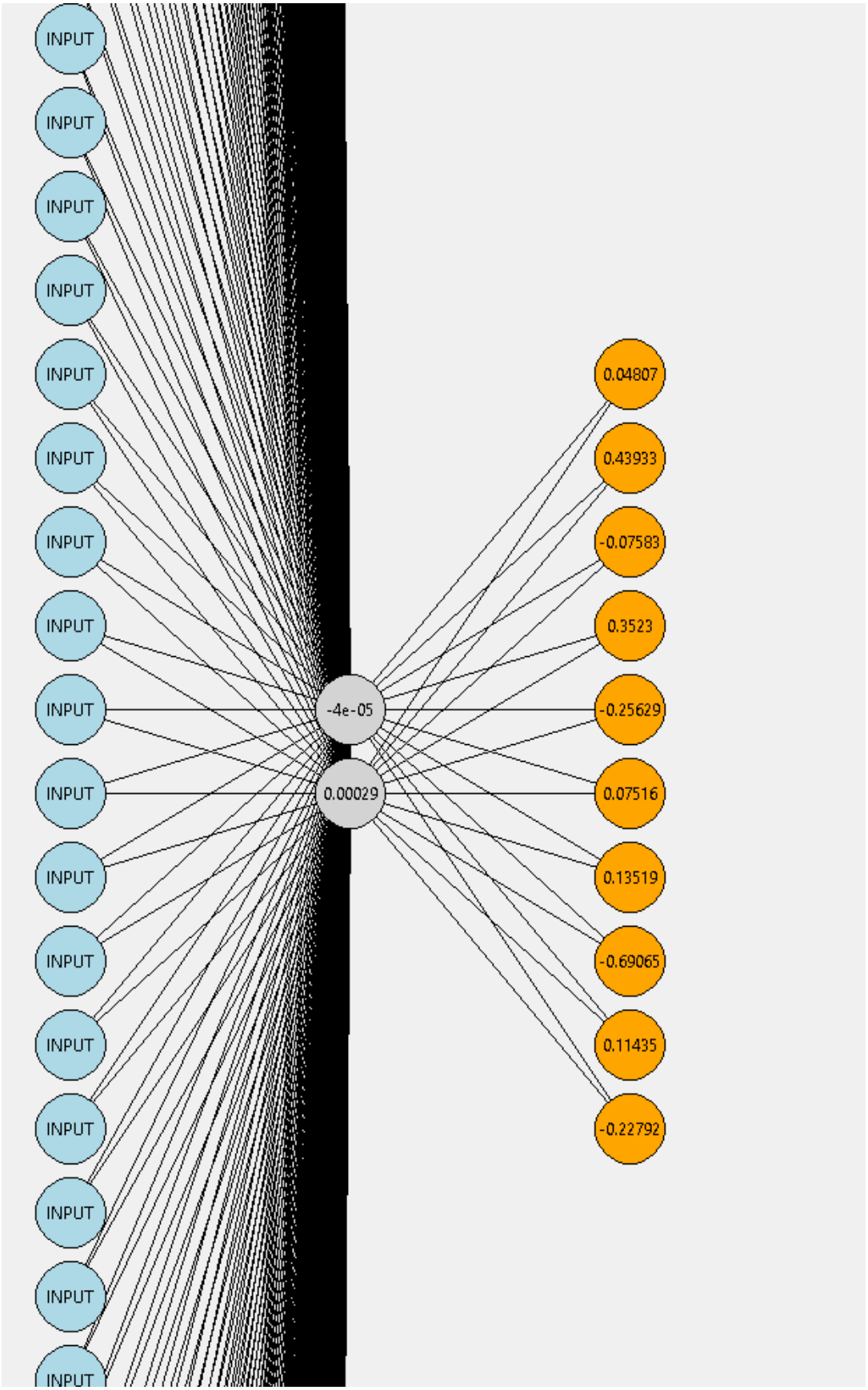


40 iterations of a model of structure [768, 2, 10] inputted



The model was predictably terrible (2 hidden nodes is nowhere near enough to classify images well).

Next I visualized it, to do so I simply opened my visualizer tool.


INPUT
INPUT
INPUT
INPUT
INPUT
INPUT
INPUT
INPUT
INPUT
INPUT
INPUT
INPUT
INPUT
INPUT
INPUT
INPUT
INPUT
INPUT
-4e-05
0.00029
0.04807
0.43933
-0.07583
0.3523
-0.25629
0.07516
0.13519
-0.69065
0.11435
-0.22792

Test 11 Screenshots:

Testing the left click in the drawing tool, as expected a black ink blot is drawn on the grid.

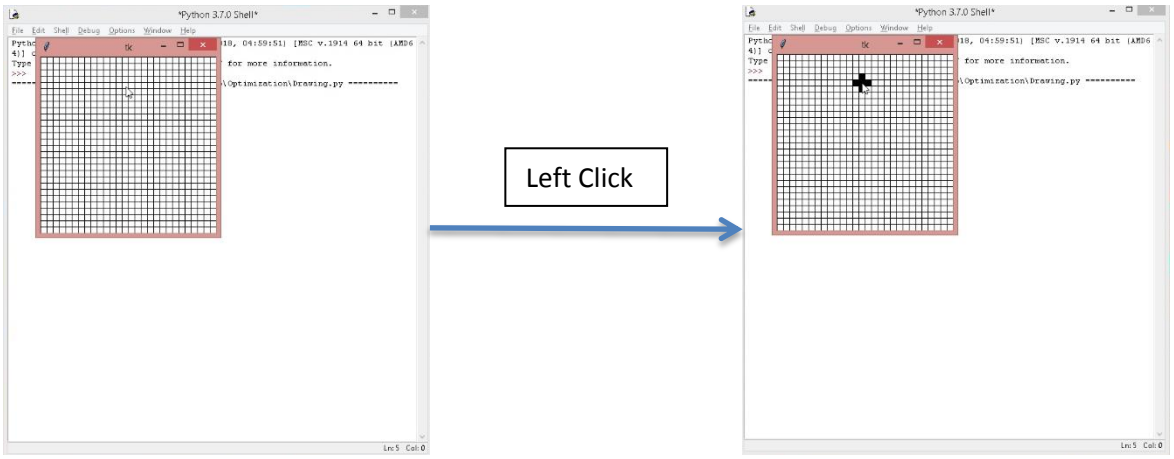
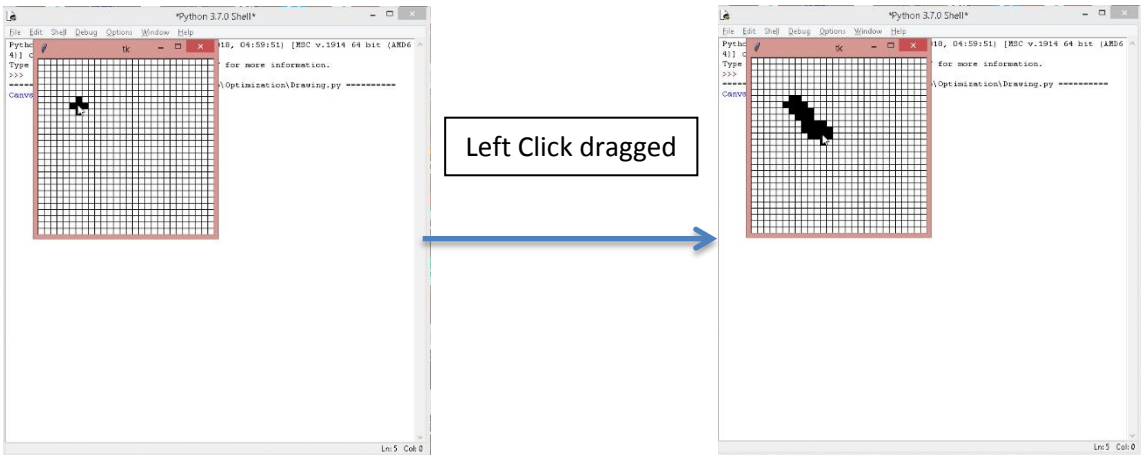


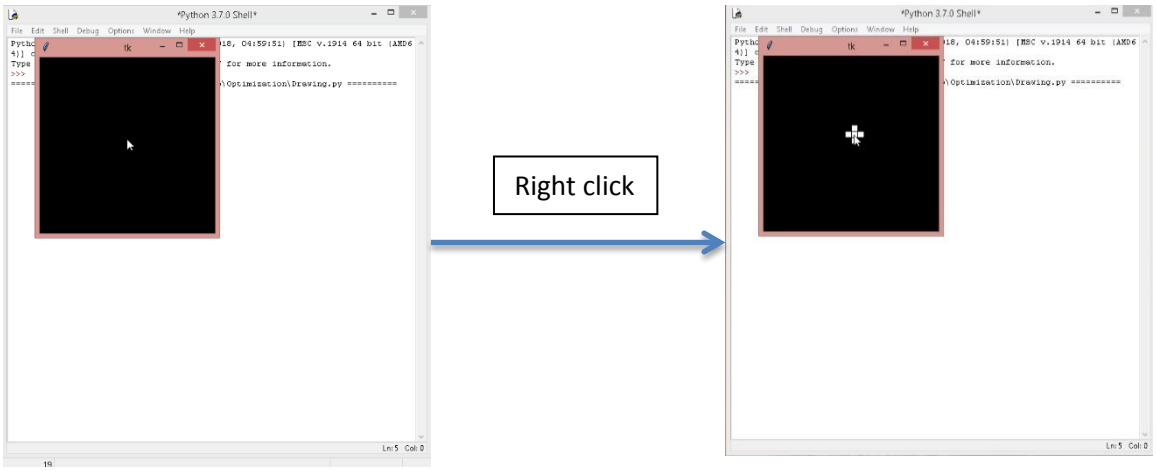
Figure 26 - The visualised network

Test 12 Screenshots:

Testing continuous drawing in the drawing tool, as expected a continuous black ink blot is drawn on the grid.

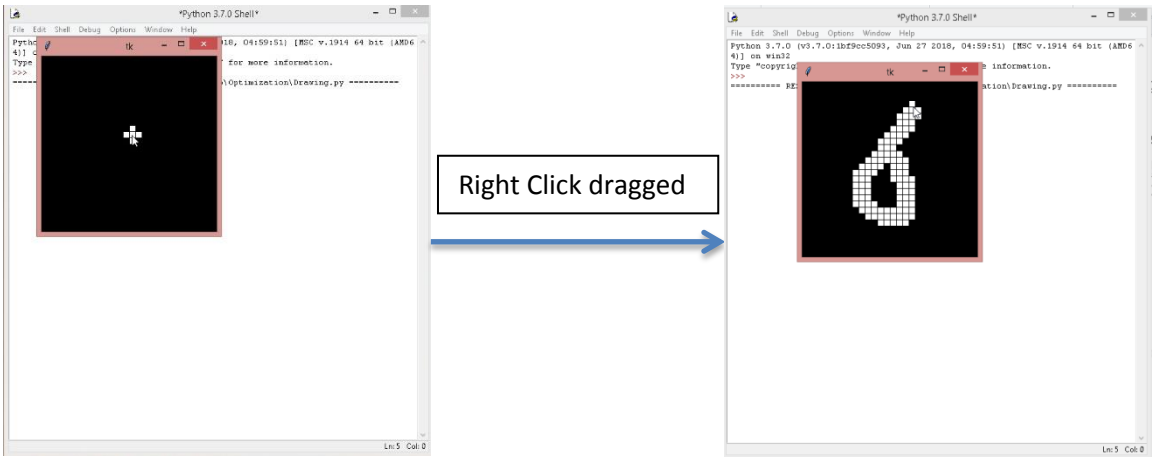


Test 13 Screenshots:



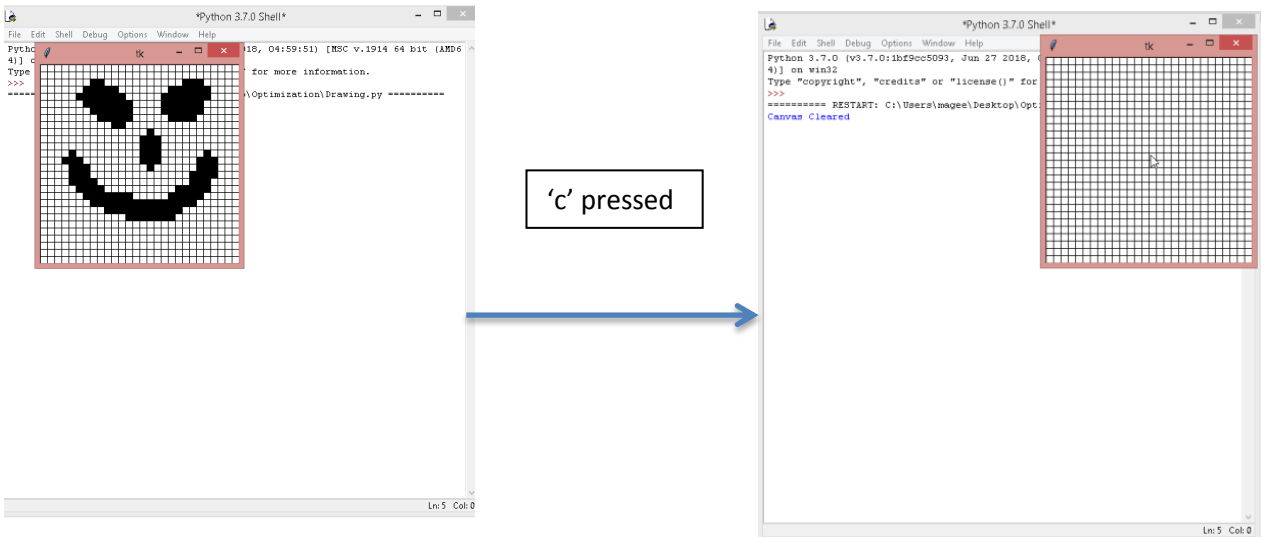
I initially filled in the grid with black, I then right clicked and the result is shown above.

Test 14 Screenshots:



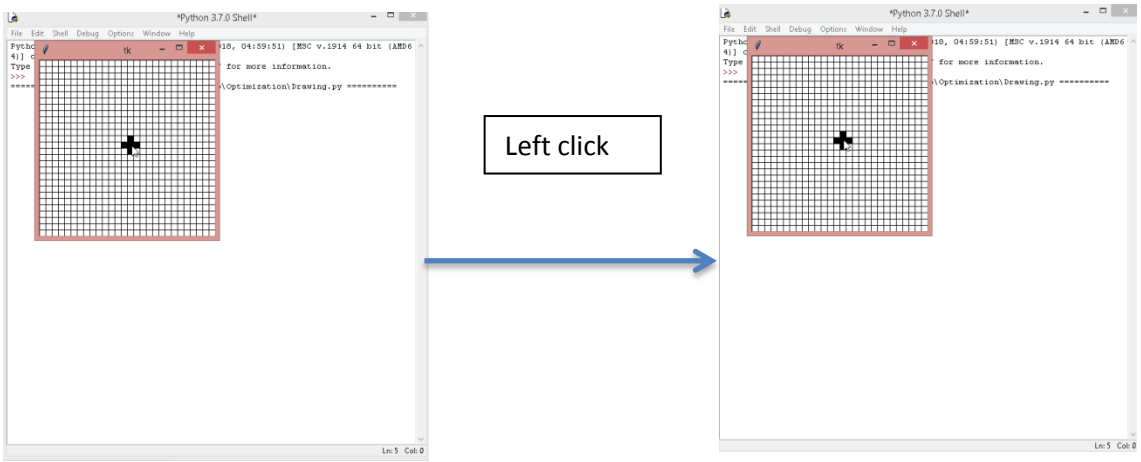
Again I initially filled the grid with black, then I clicked and dragged left click to draw a six.

Test 15 Screenshots:



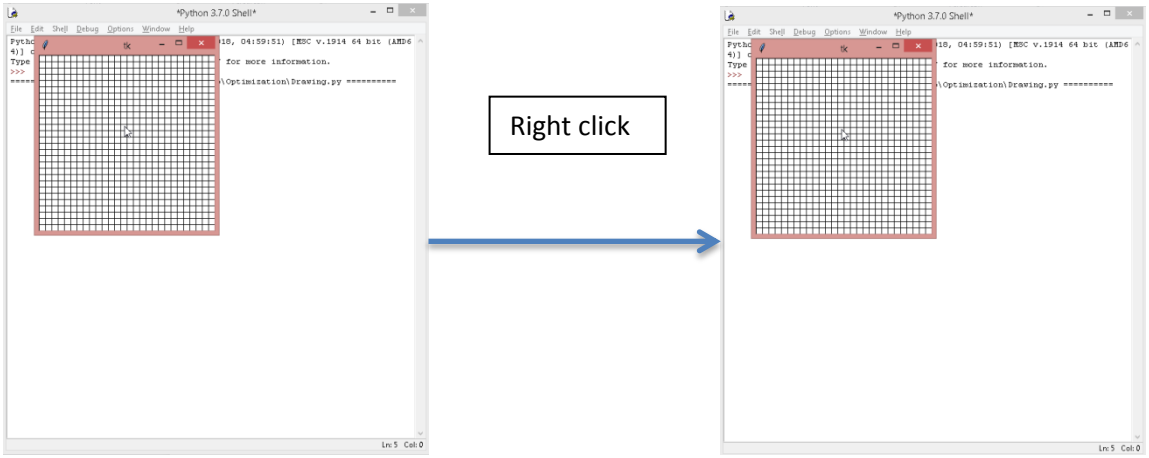
I initially drew a face to be cleared, upon pressing the 'c' key all the squares in the grid are coloured white; we are then informed that the canvas has been cleared.

Test 16 Screenshots:



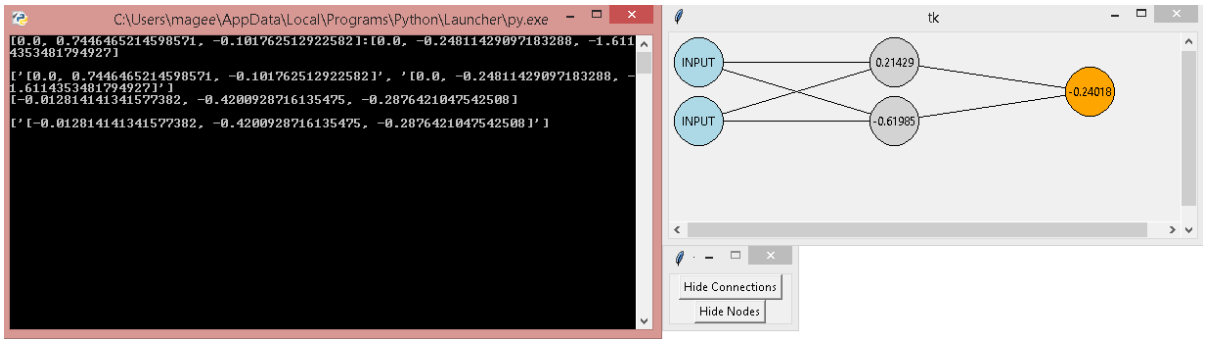
Left clicking an already coloured area gives the expected result of having no effect.

Test 17 Screenshots:



Erasing blank grid does nothing

Test 18 Screenshots:



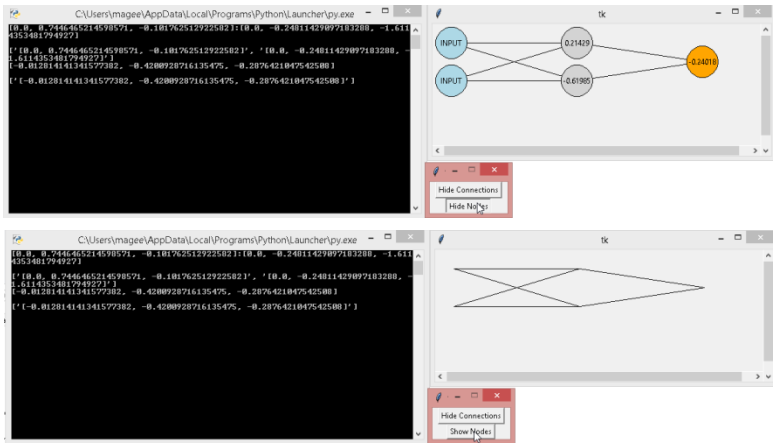
The network is visualized correctly, with input nodes marked as such and any other nodes marked with the average of their weights.

The console displays a list of the nodes & their weights.

Test 19 Screenshots:

Upon clicking the “Hide Nodes” button, the nodes disappear leaving only the connections between them showing.

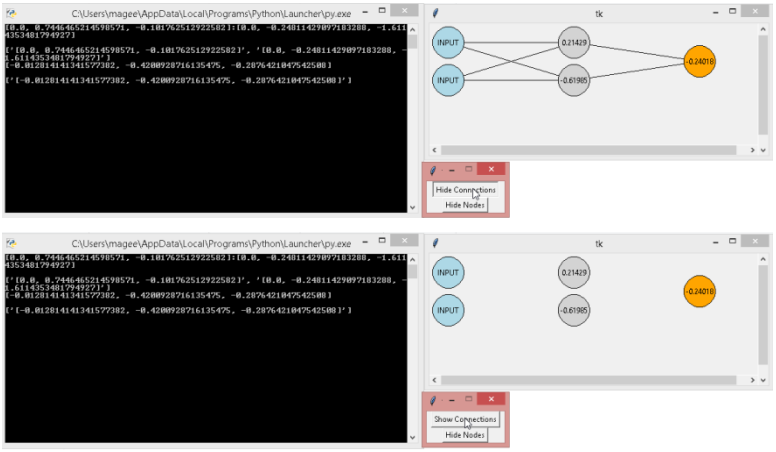
The button then changes from “Hide Nodes” to “Show Nodes”.



Test 20 Screenshots:

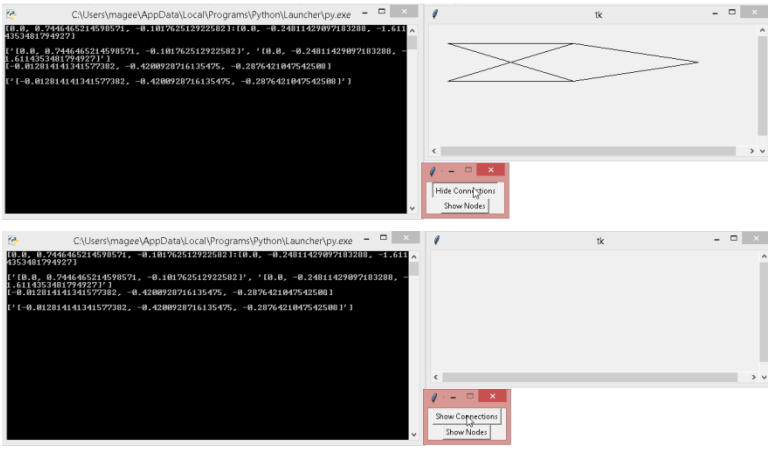
Upon clicking the “Hide Connections” button, the lines between nodes disappear leaving only the nodes and the averages of their weights showing.

The button then changes from “Hide Connections” to “Show Connections”.



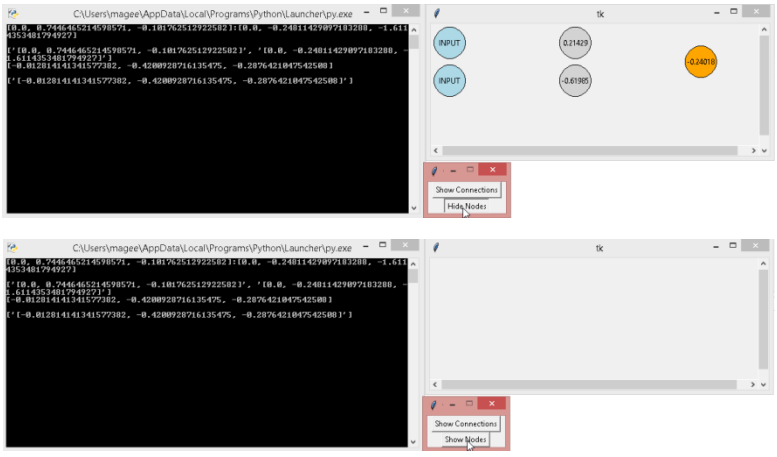
Test 21 Screenshots:

“Hide Connections” is pressed and the connections are duly hidden. All that remains is a blank screen, and our two buttons reading “Show Connections” and “Show Nodes” simply crying out to be pressed.



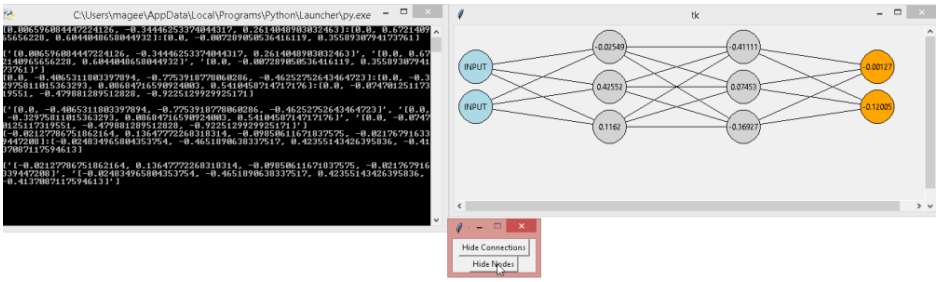
Test 22 Screenshots:

“Hide Nodes” is pressed and our nodes disappear, leaving again a blank screen and our two buttons reading “Show Connections” and “Show Nodes”.

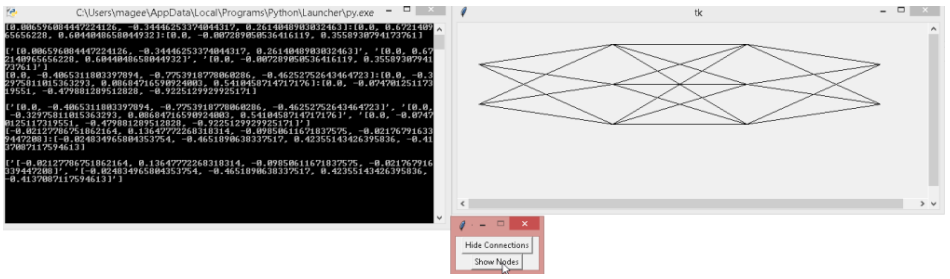


Test 23 Screenshots:

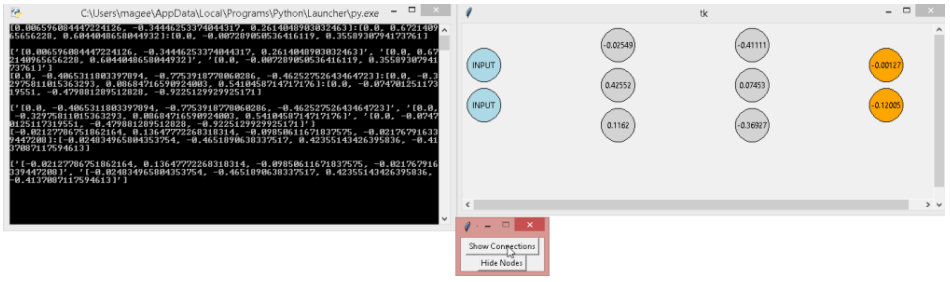
Initially the network is visualised correctly and the average of every node's weights shown.



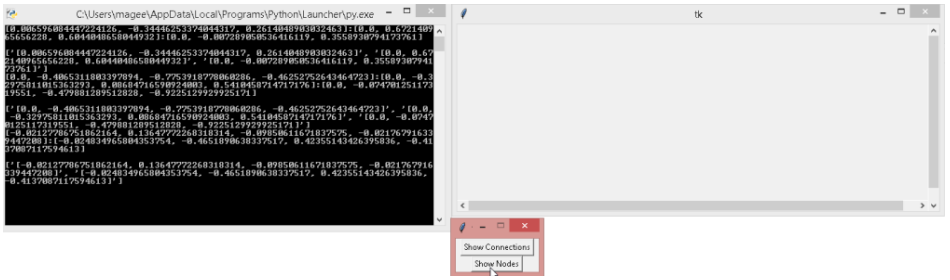
Hiding the nodes leaves only the pattern of connections between nodes.



Hiding the connections leaves only the nodes.

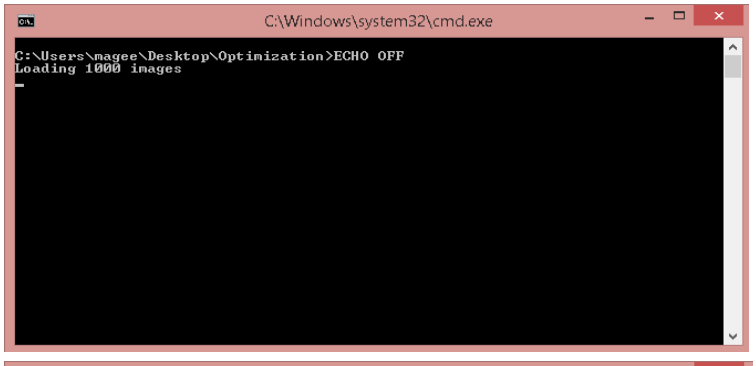


Hiding everything leaves nothing.

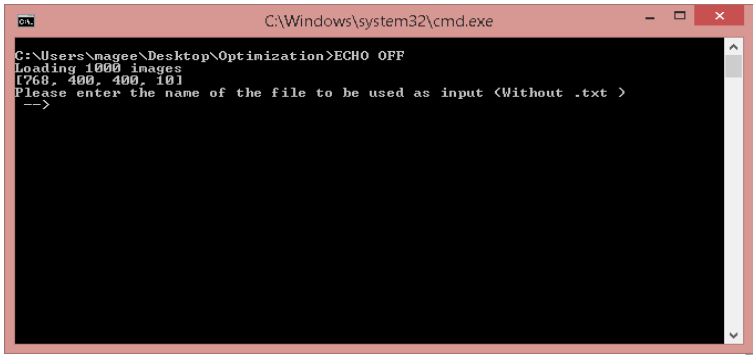


Test 23 Screenshots:

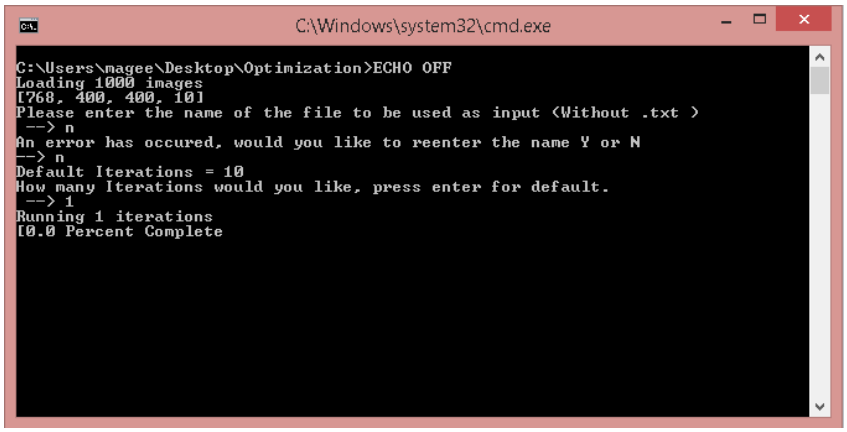
Initially the 1000 testing images are loaded.



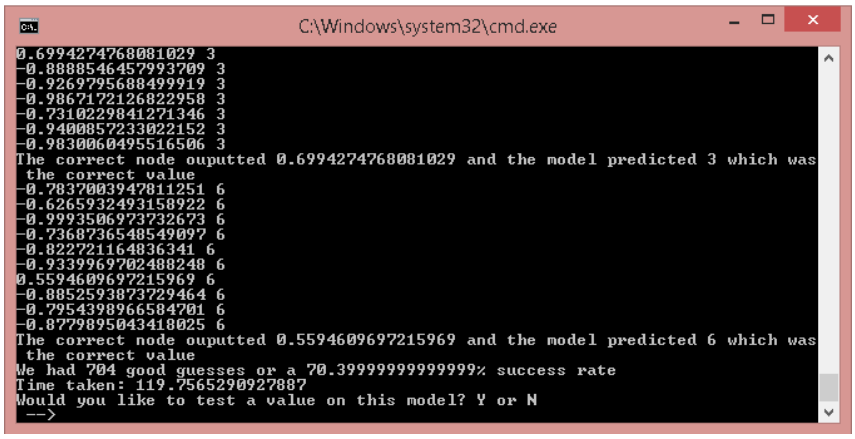
We are then asked to enter the name of our input file, as we don't want to input a file I simply press "n" twice.



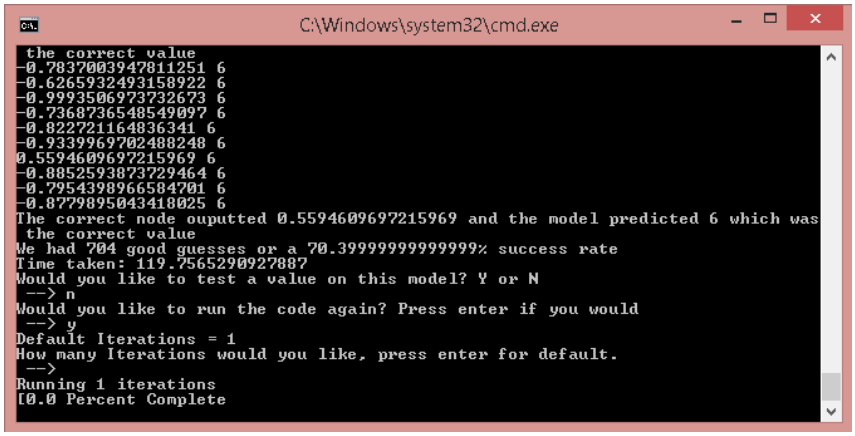
I then enter our iteration number (1) and the process of learning begins. All of this is being run in PyPy, to improve the speed of our program.



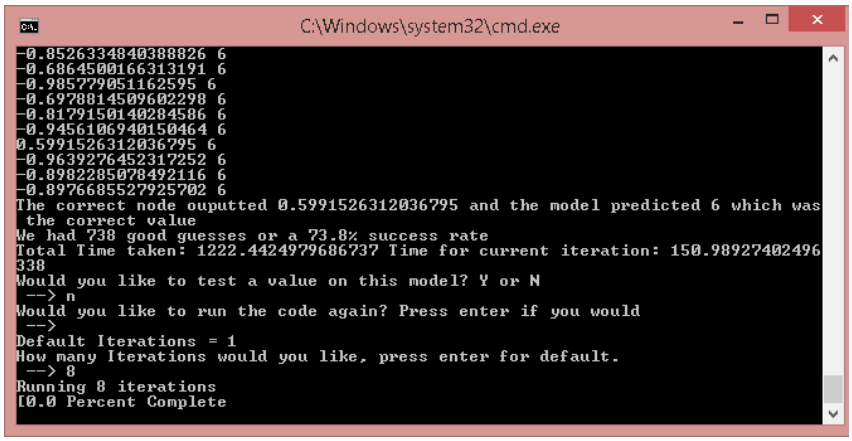
The learning process completes and the model has learnt the data to 70% accuracy, after only a single iteration. With more iterations we can achieve greater accuracy.



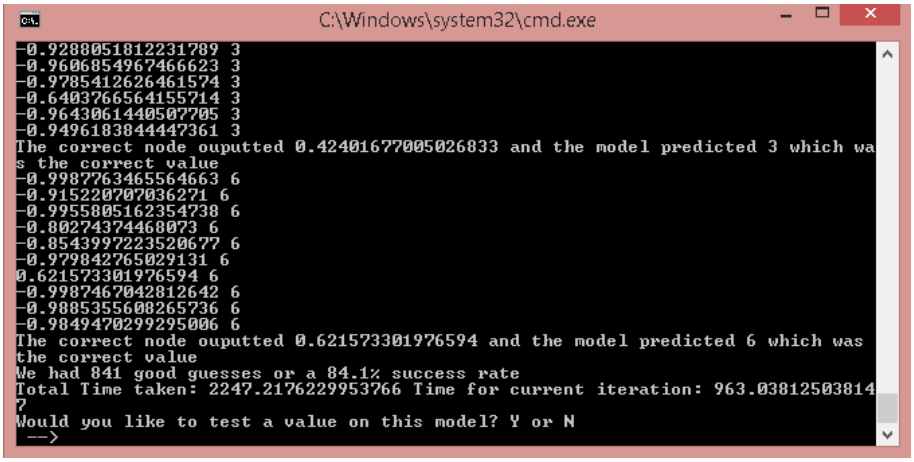
I decided to run a further iteration; these iterations are taking almost two minutes apiece, a limitation of my implementation.

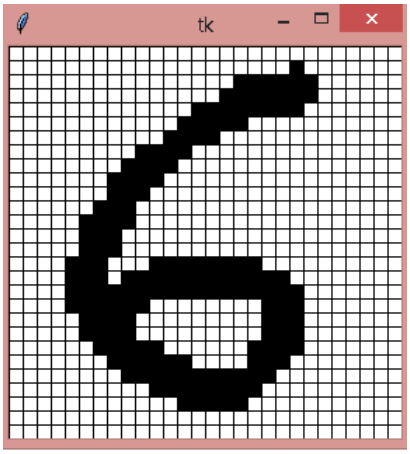


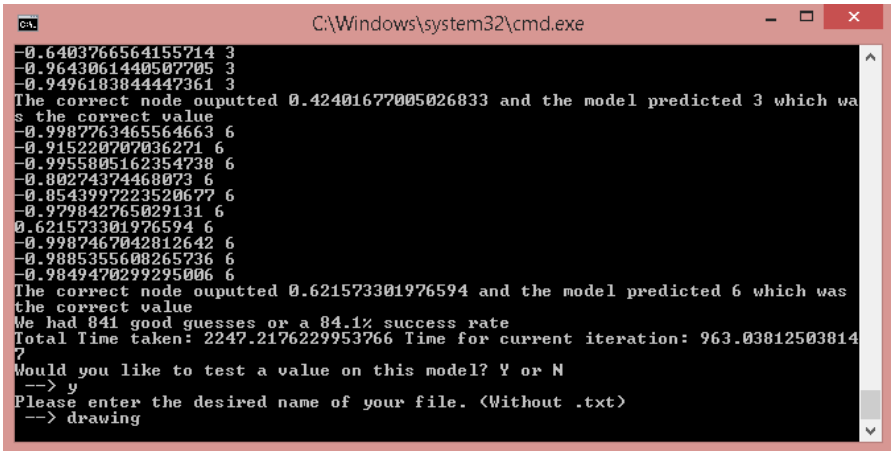
Resulting in a 4% increase in accuracy, prompting me to continue for a further 8 iterations. This should take around 16 minutes. The 1222 seconds shown here includes idle time, spent with the program open but not doing anything.



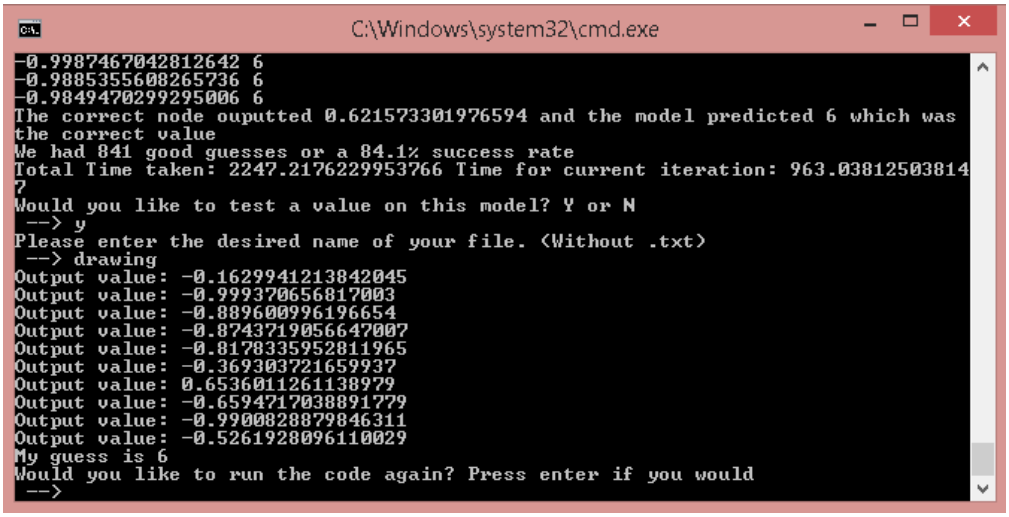
After a total of 8 inputs our model can classify our data with 84.1% accuracy. It took almost exactly the expected time and performs well. Again the total time includes idle time, in which the program is doing nothing.







To show how a normal user might interact with the program, I drew a six on my drawing tool and gave that six as input. The result is:



The model correctly predicts the value of our input. Its next best guess would have been a 1.

In testing the model, I noticed that the loading bar was occasionally giving long floating point numbers (due to a floating point arithmetic error). To combat this I simply round the percentage to two decimal places before printing it:

```
print(round(100 * i / (number_of_iterations * len(data_list)), 2),
      "Percent Complete")  # gives the percentage completed
```

Evaluation:

Objective Analysis

Objective	Met?	Comment
• Implement a system to generate and train neural networks, with the learning method being back-propagation	Yes	Upon launching the program the user finds the method of setting the number of iterations, the data to be trained from and the data to be input. The model is then trained and a solution is reached.
• Implement a linear neural network without backpropagation to show the difference between the two methods.	Yes	Simple to implement, not useful for classifying non-linear data.
• Allow the user to draw their own digits which can then be tested against a previously trained model and an output found.	Yes	The user simply draws their digit, and closes it to save. They can then load their image into their model as input.
• Allow the user to choose data from which to learn	Yes	Upon opening the program data is asked for, and can then be inputted.
• Implement a method of visualizing the network. The user should be able to hide and show certain parts of the image i.e. hide or show nodes and connections between nodes.	Yes	The visualizer works, but generalizes poorly to networks of a large size. When you have 768 input nodes the result of visualizing is ugly and hard to understand. However the averaging of weights is interesting and gives some understanding as to how the model works.

Possible Extensions:

Were I to complete this program again there are certain changes I would make:

- Learn and write the program in a compiled language
- Use Numpy and other machine learning modules (Numpy isn't specifically for machine learning but most machine learning implementations in Python use it)
- Implement different methods off machine learning that can operate on non-linear data
- Implement a more streamlined and visually pleasing GUI
- Implement multi-threading

I would learn a compiled language (to the level of familiarity that I have with Python) simply due to runtime. To train a suitable sized neural network with 768 input nodes, two hidden layers of 400 neurons each and an output layer of 10 neurons takes 8 hours when training on the entire MNIST dataset. And to reach a sufficient level of accuracy of over 80% this had to be repeated for a further 8 hours. The speed of Python for computation (even using software like PyPy) is not to die for.

But if I had to write the program in Python again I would definitely use modules like Numpy or Theano. These modules can perform operations that are run thousands of times in my program, blazingly fast. While I chose to write my implementation without using the machine learning modules, they would have been incredibly helpful both for implementation and runtime.

While I have implemented both a linear neural network and a network that uses backpropagation, were I to complete the task again I would also implement k-nearest neighbours, a genetic algorithm or any other machine learning algorithm I chose. As I've repeatedly mentioned, different machine learning solutions have different strengths and weaknesses. To be able to show that by giving the two implementations the same data and seeing the variation in outputs would be fascinating.

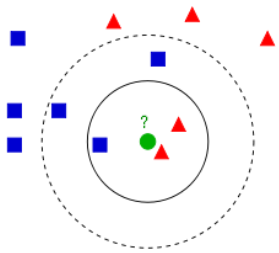


Figure 27 - Example of K-Nearest neighbours from https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

On top of that visualizing the two networks side by side would be great for improving the understanding of others.

I would also implement a more visually pleasing way of interacting with the program. At the moment (using Python) the command line of PyPy was necessary for taking input, however a different implementation could have a Graphical User Interface for inputting and outputting data. Additionally the visualization of the network could be made more visually pleasing.

Multi-threading would also be a priority for implementation were I to complete the project again, as this would greatly improve the run-time of the program. Multi-threading is a technique in which a single piece of code can be used by different cores of the processor at different points in execution, rather than simply using one core. While my PC at home has only 1 core in its processor, more modern PCs can have many more. I would also like to make use of the Graphics Processing Unit in my home PC, as calculations in machine learning are often well suited to the GPU.