Detection of Parkinson's

by virtue of Neural nets using mouse data

Summary

For my investigation I am looking to detect symptoms of Parkinson's from mouse data. Parkinson's is a neurodegenerative disease in which people lose mobility and begin to experience tremors, spasms I their muscles that they cannot control. I hope to investigate whether these tremors can be detected from vibrations in mouse movement during day to day use on a computer by running a program in the background of PC's belonging to those with the condition.
I intend to collect data by recording mouse movement and sending it to a central server for analysis and storage in a database. I intend to contact people with the condition in my local area/ a Parkinson's society to be granted permission to install my software on their computers. From there I hope to train an AI to recognise tremors caused by Parkinson's the data set collected containing, people with the condition, and general mouse data sourced from running my program in college or using data sets online.

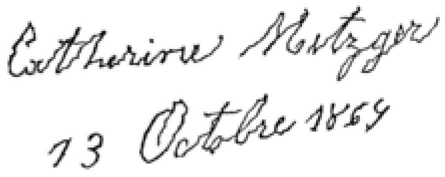
This subject particularly interested me due to its real-world applications and for giving me the opportunity and motivation to delve deeper into the realm of machine learning, networking and development in for the windows environment. I am intrigued as to which form of ML will be best suitable for my scenario and the implications involved in implementing the most efficient solution, in terms of optimisation and mathematical challenge. Additionally, I have never incorporated networking into a solution, it will be interesting to see the options available to me with my chosen language C++.

Contents

Initial research

Parkinson's

Parkinson's disease, denoted PD from here on, is a currently untreatable degenerative disease that affect the central nervous system, that mainly affects the motor neurone system, the nervous system responsible for controlling movement. The symptoms develop slowly with shacking, rigidity and slowness of movement and later thinking and behavioural issues develop. Dementia, depression and anxiety are common with one third with PD having such issues. The main motor symptoms are denoted "parkinsonism". The stimulus for the disease is unknown, but are believed to be contributed by genetics and environmental factors. Parkinsonism generally has no identifiable cause with intense periods occurring randomly in the subject's life. The most common occurring sign is a harsh slow tremor in either hand at rest which disappears during voluntary movement of the hands or arm and in the deeper stages of sleep. It typically appears in only one hand, eventually affecting both hands as the disease progresses. Parkinson's differs from other neurodegenerative diseases as it also causes an abnormal accumulation of alpha-synuclein protein in the brain unlike Alzheimer's where tau protein accumulates. "Considerable clinical and pathological overlap exists between tauopathies and synucleinopathies" in laymen's terms this means that it will be very difficult for me to distinguish between Alzheimer's and PD since their symptoms are very similar although the sampled data can be filtered to contain only parkinsonian mouse movement which may allow the chosen algorithm to recognise and ignore Alzheimer's, as an extension I could allow the algorithm to recognise Alzheimer's although that would require me to collect a 3rd data set and be out of scope of my investigation. Memory loss is also a key component of PD and becomes very prevalent in the later stages of the disease, I will have to take this into consideration when developing the project such that the user doesn't have to remember any details required for the collection to be successful; I believe that this should extend to the so


Catherine Metzger
13 Octobre 1869

Handwriting of a person affected by PD

Summary

Symptoms are most obvious when the hands are stationary although, early symptoms occur only in one hand, and could therefore be easily missed until the condition worsens.

Symptoms / detection

Neurodegenerative diseases progress slowly with subtle symptoms at first but early diagnosis can slow their progression. PD causes tremors that are Unilateral and typically occur between 4 and 6 Hz any Hand Tremors are supination–pronation ("pill-rolling") tremors that spread from one hand to the other. PD causes rigidity in the subjects' limbs and tremors result in a cog-wheel motion. It can also induce voluntary finger tapping which I could detect in mouse data, since I have no way of deducing whether the movement was voluntary or not I don't believe it necessary to record mouse clicks too especially since I must minimise the data collected and transferred. Another symptom is slowness in making voluntary movements, this could be shown in the modelled data by smoothing the source data initially then applying the 4-6Hz sine wave. Due to the constant tremors causing increased muscle tone to the rapid contractions in the patients arms would lead to sharper peaks in the mouse movement with the involuntary movements being severe, this would again lead to noise in the data but knowing this I will scale the noise more such that it affects the data more severely. It has also been found that PD is often correlated with smaller handwriting, this could lead to generally smaller mouse movement with the sharp peaks in between.

Previous studies found it difficult to distinguish between PD and essential tremor (another nervous disorder causing similar tremors), but were able to use a multitude of machine learning classification models which were trained on two distinct groups to successfully classify a new sample with 96% accuracy far better than the 75-80% accuracy of clinical diagnosis. A later study found that when recording the tremors with an

accelerometer, an FFT could be applied to the data (Fast Fourier transform – converting amplitude time signals to the amplitude frequency domain) making it was easy to distinguish traces of PD in a subject due to clear differences in the frequency of oscillations in the subjects' hands.
(Ryen W. White, 2018)
(Jankovic, March 14, 2008.)
(Adams, November 30, 2017)

Summary

Two groups of mouse data should be collected, that of PD sufferers and non-PD sufferers. An FFT should be applied to the data, spikes at a frequency between 4-6hz should be visible. Classification model should be applied with a target accuracy greater than 75%.
If a model is to be used: since healthy mouse data will be easy to collect I will first smooth a collection of healthy data to represent the slowness of movement then will apply an artificial 4-6Hz sine wave with noise.

Scope of the Problem

There are many elements of the project which require research the first of which is the feasibility of how to recognise PD from data, and the best machine learning algorithm to do so. Next is the ability to collect mouse data from windows machines and whether this data will be detailed enough to draw sensible conclusions as well as collecting time stamps for the mouse data. Finally, I must research how the PD mouse data will be sent to me for storage and how I will be formatting and storing the data.

The issues with detecting Parkinson's

The key issues that arise when diagnosing PS are that the symptoms progress linearly over time this makes initial diagnosis very difficult as the symptoms are very minor, due to this, doctors test numerous factors that suggest PD but no one test can diagnose the disease. A combination of your medical history, since PD is often hereditary, a specific single-photon emission computerized tomography SPECT scan called a dopamine transporter (DAT) scan can be used to support the fact that you may have the disease but cannot prove it. Imaging techniques such as MRI scans can also be used but moto symptoms are most prevalent.

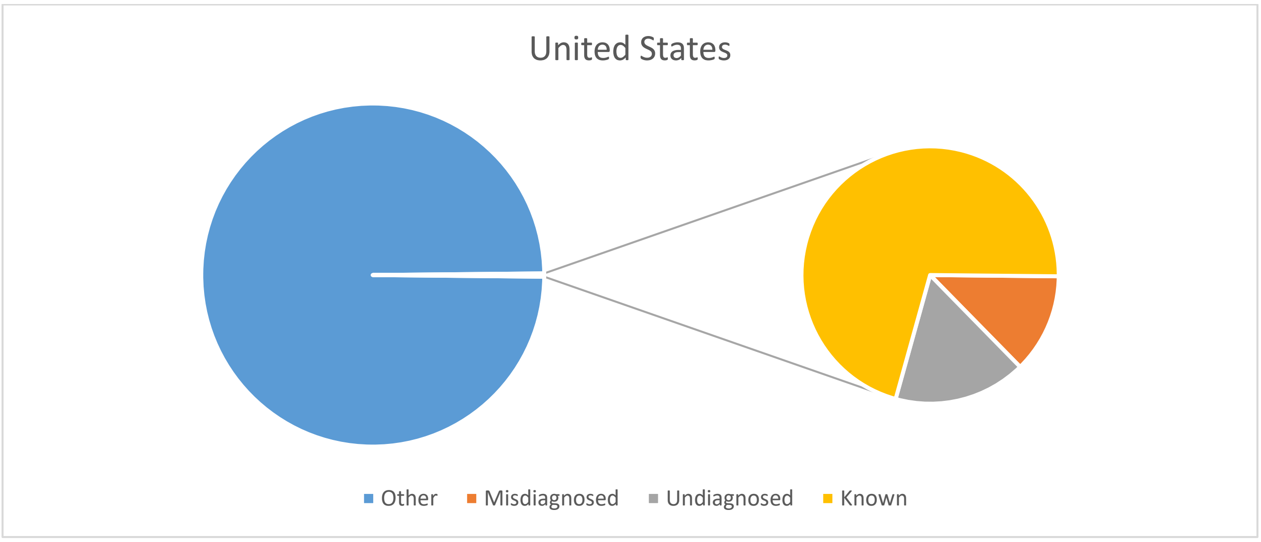
Classification

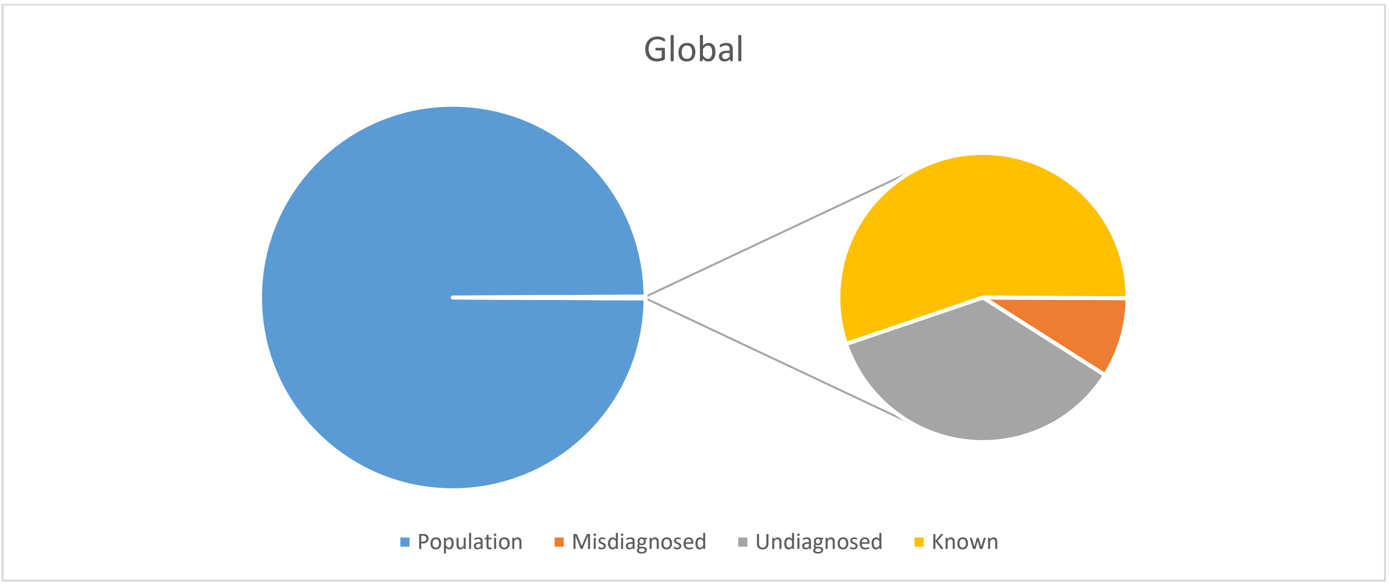
I will be classing PD as the following:

- Tremor at 4-6Hz
  - This will take shape as a circular motion with a period of 0.166 to 0.25 seconds
- Generally slow movement
  - This will take shape as smoother movement towards a target with mean velocity being less than normal mouse data
- Random Spikes of quick movement
  - These will appear as spikes in movement occurring randomly with varying severity
- Generally small mouse gestures
  - Would be shown by mouse movement being restricted to smaller areas within the screen, ie the range of the co-ordinates will be less than average

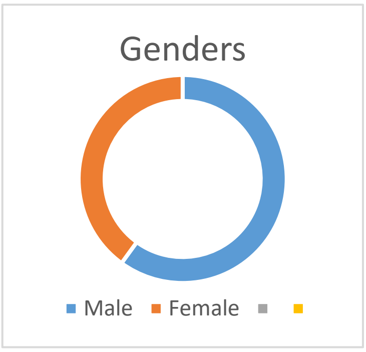
Statistics

It is estimated that nearly one million people will be diagnosed with Parkinson's disease in 2020 in the United States alone, since the current population prediction for the United States in 2020 stands at 389 million it follows that 0.257% of people in the united states will be diagnosed by the year 2020, affecting the lives of many more, up to 10 million people worldwide are thought to be living with the disease WITH ONLY 6.1 million cases bing known. Every year approximately 60,000 people are diagnosed additionally, 15% of people are misdiagnosed with Parkinson's, failing to meet the strict clinical criteria for the disease and then 20% of people with Parkinson's evaluated were not diagnosed despite meeting the aforementioned criteria. I have concluded these results below:





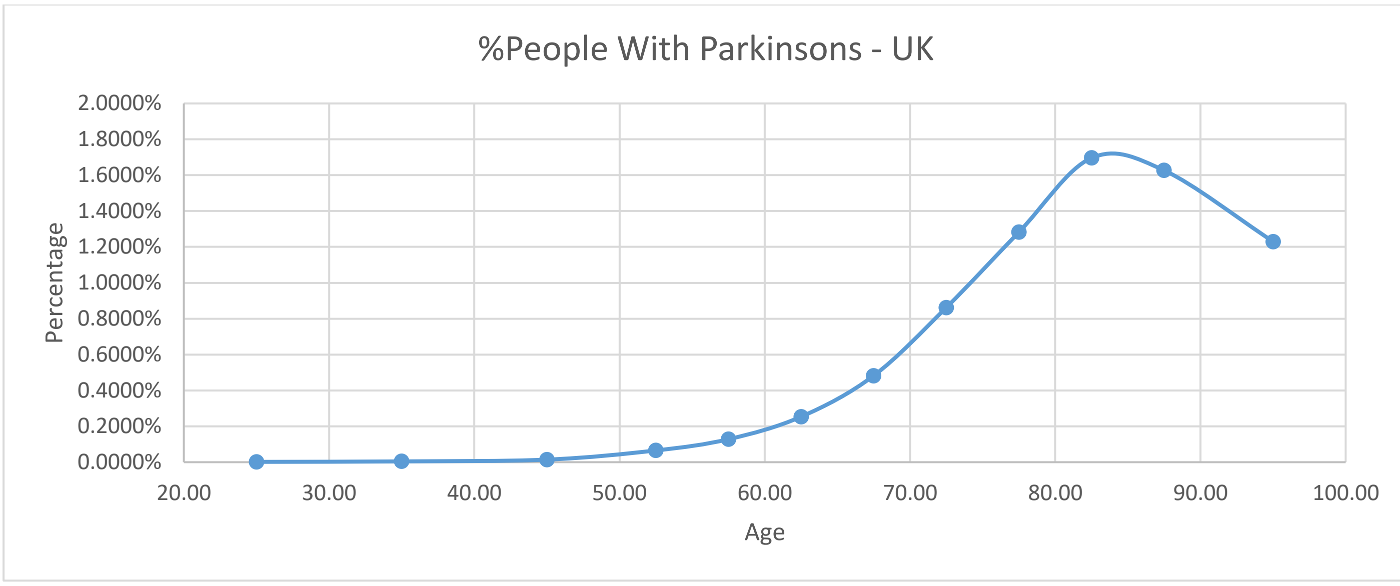
Gender Distribution



Men are 1.5 times more likely than woman to contract the disease but it is not known why, although it is currently thought be a combination of the protective effect of oestrogen, the increased rate of minor head trauma in men, as well as the, on average, increased intake of toxins by males, finally the increased genetic susceptibility of male chromosomes.

Age Distribution

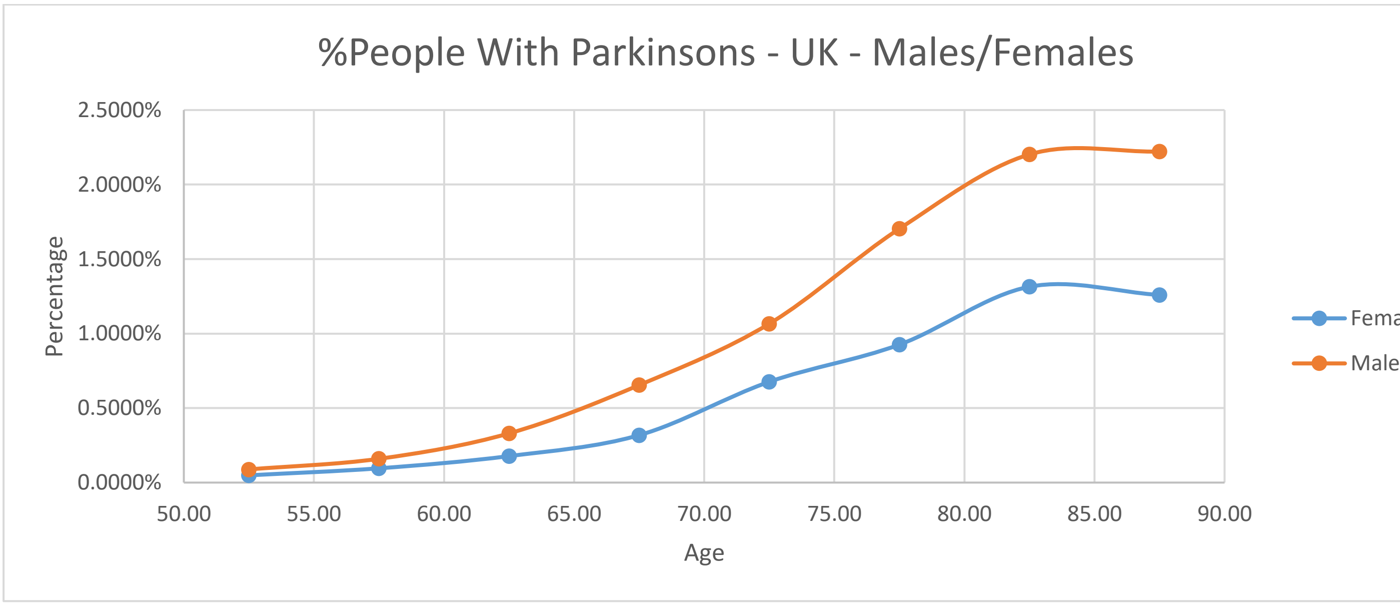
As people age their susceptibility to developing the disease drastically increases, for those in their 40's just 67 people out of 471,034 contract the disease, that's just 0.014%. By the time you're 80, 2318 in 138874 people contract it, that's a rise to 1.7%! The full breakdown is as follows: 20-29: 0.002%, 30-39: 0.005%, 40-49: 0.014%, 50-59: 0.194%, 60-69: 0.736%, 70-79: 2.145%, 80-89: 3.323%, 95: 2.857%.



As you can see, once people with the condition pass 85 years of age the percentage with Parkinson's begins to decline, this is due to the high fatality rate of people with the disease in later stages.

Age Distribution in each gender

As you can see the graph demonstrates the increased susceptibility of males vs females.



(Parkinson's UK, 2017)

Interview with 3rd party

Initial discussion:

Johnathon: So, your trying to distinguish between parkinsonian like tremor compared to known non-parkinsonian tremor in students, what do you think will be the most difficult aspect of your project?

Me: I believe it will be a sufficiently accurate training a model to distinguish between the two models, as that incurs a lot of complicated maths and statistics within computing.

Johnathon: A problem I thought of was the difficulty in obtaining the Parkinson like data how do you aim to tackle that issue?

Me: I have proposed my project to the Parkinson's society who haven't responded, I have also researched the local area and had very little response. I believe the best method of collecting parkinsonian like data will be to create a model of parkinsonian data, incorporating a 4-6hz sine wave as that was shown in previous research to be a key component of the tremor.

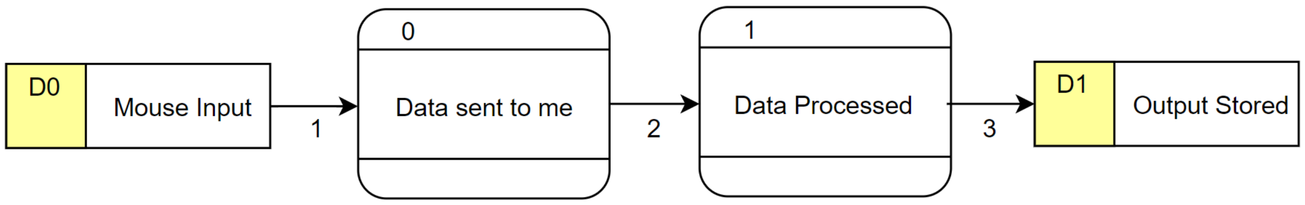
Yes, I agree that you need a model given the difficulty of finding people with Parkinson's because I doubt the Parkinson's society will really respond since there is nothing in it for them, especially given that Parkinson's is a very well recognised disease with much research conducted previously over the years. The other issue is that people who have the disease under medication will have diminished tremors and will be very difficult perhaps to correlate tremor against drug dosage.

1. Do you think It will be possible to recognise Parkinsonian behaviour with machine learning, if so are there any implications in doing so, that come to mind?
    a. I definitely think you will be able to do your machine learning to distinguish between the normal non-tremor state the simulated tremor state. Quite how that will genuinely relate to Parkinson's itself might be more difficult and would need cooperation from Parkinson patients and compare the two data sets.
2. Are there any data protection issues I should be weary of?
    a. I think that's the main reason why the Parkinson's society would be reluctant to comply as they will be unlikely to go ahead with that for all sorts of data protection act issues.
3. How applicable do you think my investigation is to the real world?
    a. I honestly don't know as I think that, other than the machine learning component which I know you will do, if you take the comparison of the parkinsonian like tremor against real Parkinson's tremor would be a study in itself and therefore be out of scope of your study. Therefore, I think that you should not refer to it as detecting Parkinson's disease, you should always refer to it as detecting your normal mouse movement compared to your modelled parkinsonian moments with a 4-6hz sine wave applied.
4. What symptoms do you think will be the most obvious?
    a. I think there is a whole symptomology in Parkinson's such as the tremor and even gait. But I think all you can do is to try and calibrate your model, if you were lucky enough for someone with the disease to collaborate also while not under medication, without which you can only detect Parkinson's like tremor
5. Do you believe my solution could be used for clinical diagnosis? If so would there be any issues in doing so?
    a. Its certainly out of the current scope of the current project but if data sets where very clear it would be more feasible to contact the Parkinson's society once that data had been obtained.
6. Do you believe there is any merit in using alternative methods of detection such as eye tracking / neurological detection?
    a. I would have thought that university groups would be certainly looking in these areas
7. Is it a good idea to display results should there be an optional consultancy?
    a. If you get a clear-cut result that is proven to distinguish between the data sets, I think it would be very interesting to pass on the results to a clinical environment
8. Do you think environmental factors such as medication and diets could affect the results? If so how could I take precautions against them?
    a. You could in your control group experience tremors from other sources such as alcohol. Consequently, you must have a very large data set to reduce the affect of these anomalies. Equally in normal healthy use its plausible that people show similar tremors or could have the symptoms of other tremor inducing diseases.

Interview Summary

It is very important to develop as accurate of a model as possible possibly including a 4-6hz sine wave to represent parkinsonian behaviour and consequently must be referred to as parkinsonian like data. This is due to the fact that Parkinson societies are unlikely to corroborate with me due to DPA issues and the fact that people will not want to share their mouse data. It would be interesting to pass on the collected data for future research. There will probably be signs of parkinsonian behaviour in the healthy data.

Level 0 data flow diagram



Arrow ID	Name	Description
1	Compilation	Mouse inputs written to a file with time stamps which to then be sent to me
2	Decoding and pre-processing	Data normalised and passed to Machine learning algorithm
3	Results	Results collected from machine learning algorithm and sent to output store

Data Analysis

I first must establish which machine learning technique I will be using to analyse the data I will be collecting; for this I found some useful sites, I have summarised key points from each.

Different types of Machine learning

Supervised:

A large set of training data is used to create "logic" that can be applied to similar data. "Fast and accurate". Used to "model relationships and dependencies between the target prediction output and the input features"

Nearest Neighbour

- Naive Bayes
- Decision Trees
- Linear Regression
- Support Vector Machines (SVM)
- Neural Networks

Unsupervised:

A larger set of unlabelled training data, forcing the algorithm to learn patterns and relationships within the data on its own, clustering the outcomes.

Reinforcement:

Trial and error approach to solving the issue, eg genetic algorithms, making the algorithm change and run again discarding negative traits

Semi-Supervised:

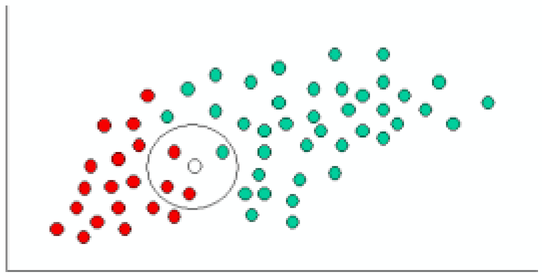
Only some of the data used is labelled the un-labelled data is then grouped according to the labelled data, combining the learning technique from unsupervised and supervised learning

Conclusion:

From my research I believe it is clear that supervised learning is the most method of machine learning in investigation since labelled data collection is easy; furthermore, the data can only be categorised into my labelled types (Parkinsonial and non-Parkinsonial, both are mutually exclusive and combined are exhaustive).
(Sanjeevi, 26 September 2017)
(Fumo, 2017)

Different types of Supervised Machine learning

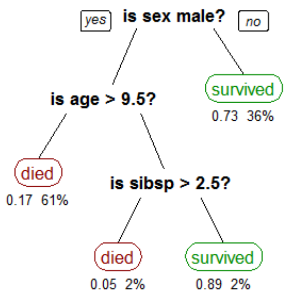
Naïve bays
Probabilities of each outcome calculated by assuming they are independent, that's all that is done to train the AI, data should be Gaussian or near-Gaussian (fits a bell curve).
(Brownlee, 2016)



Eg there are 20 red, 40 green, in the vicinity of the new (white) cell there are 3 red, 1 green.
P(X) | Green = 1 / 40 = 0.025
P(X) | RED = 3/20 = 0.15
⇨ Most likely red
(Dernoncourt, 2016)

Decision Trees (Classification type)

Probabilities of each outcome calculated at each condition test of the data, the more conditions the item is tested for, the more accurate the result. The calculation of the conditions (splitting of the tree) is how the algorithm is trained. An example of which is **recursive binary splitting** where all possible conditions are tested and the ones with the lowest cost are chosen (I think this will take too long given my data inputs). The root is always the biggest factor in the decision.
(Gupta, 2017)



Linear regression

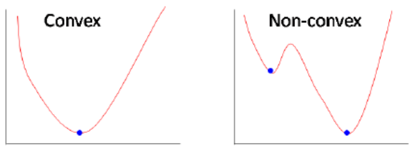
Linear relationship between input(s) and output, (applicable to Parkinson's as it linearly gets worse). It works by applying a linear equation to each input variable, with a coefficient and a constant; the model is trained by estimating the coefficients and improving on them over time. The two common techniques are "Least squares" and "Gradient Decent".

Least Squares

Draws a line of regression through the data and attempts to minimise the distance of all the data points to the line of regression by summing the square of all the distances to the line of regression. Mean Squared Error (MSE).

Gradient decent

Start with randomised values for each coefficient and iteratively minimising the error of each coefficient by increasing or decreasing it to reduce the error between the output and the labelled result, the iteration is continued until there is no improvement or a target is met. This method is susceptible to non-convex data as there could be multiple minima for each co-efficient.



After both techniques each small coefficient is reduced to zero provided it does not affect the output by a given amount, improving the efficiency of the model.
(Brownlee, 2016)
(Gandhi, 2018)

Support vector machines

One of the most popular machine learning algorithms. Allowing an n-dimensional input (applicable to my project as there will be as many mouse properties recorded as possible.

Maximal Margin Classifier

Taking a set of n-dimension points and drawing a plane that best separates them, called a "hyperplane", the equation for the hyperplane is calculated by maximising the margin, the distance from the hyperplane to the nearest points, thus maximising the accuracy of separation of each class (although only working for binary classes).

Soft Margin Classifier

Since most n-dimensional data cannot be perfectly separated a C parameter is introduced which allows for an amount of error, allowing points to cross the hyperplane up to an extent while training the model. The greater C the less sensitive the model is to the training data.

This technique is developed further with the use of kernels representing the dot product between new data points and the hyperline, the model is then trained using stochastic gradient decent for efficiency.
The previous two methods do not seem applicable to my scenario as the co-ordinate data will not be split due to Parkinsonial behaviour as although it's a binary classification the developmental stages of the disease are not binary.
(Brownlee, 2016)

Neural networks

Feedforward Neural Network

Data travels in one direction through each perceptron, where weights are applied to the input, if the output is >=0 the neurone will fire feeding into the next layer of neurons. Each neurone is trained using the delta rule, a form of gradient decent. These are applicable to noisy data and are easy to maintain. With multiple layers can approximate with arbitrary accuracy any periodic function.
(Wikapedia, 2018)

Radial basis function Neural Network

A two-layer neural network that uses a radial method of analysis, viewing the inputs as points on a circle in relation to its centre. The radius of the point to the centre can then be used to predict that the next point will probably have a similar radius. However, this method is often used to analyse risk of successive events such as failures in a power grid, consequently I don't believe its applicable to my scenario.

Kohonen Self Organizing Neural Network

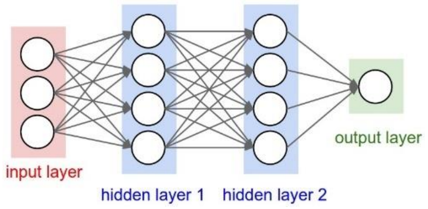
Maps a n-dimensional data space to a one or two-dimensional map comprised of many neurones. To train, each data point is iteratively selected and the closest neurone to it is chosen, it is then moved closer to the data point and the connected neurones are also moved closer according to the Euclidian distance. This method of training and operation is typically used to analyse images for a given trait.

Recurrent Neural Network

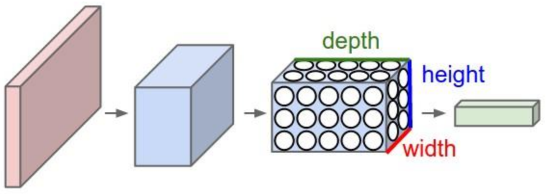
Much the same as a multi-layer feedforward neural network although each neurone will be fed some information from the previous layer of neurones. At each stage the prediction of that layer is calculated and compared to the correct output. The previous layer can then be adjusted as to get as close to the correct prediction as possible, thus reducing the error, this is called backwards propagation and is repeated until the network converges to a predefined target error state. I believe this "long short-term memory" will aid the accuracy of the neural network in my scenario as well as the ability to input an arbitrarily long sequence of data. Furthermore, according to the paper linked below, this type of neural network is capable of recognising handwriting, and therefore I believe is fitting for my scenario.
(Alex Graves, 2008)
(Wikipedia, 2018)

Convolutional Neural Network

Similar to multi-layer feedforward network although the data is inputted in batches. Inspired by animal visual cortexes such that each neurone is not connected to every neurone in the previous layer vastly decreasing the number of required weights, this would be useful in my application since I intend to collect large swaths of data and therefore having an optimal analysis technique would be preferable but CNN's are usually only optimal for image processing. Additionally, according to the stack exchange, it can only have fixed length inputs.



Standard 3-layer neural network



Convolutional Neural network in 3 Dimensions

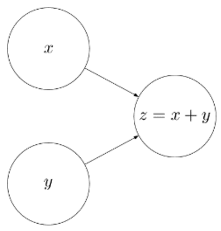
(Wikapedia, 2018)
(Freeman, 2016)
(Maladkar, 2018)

Conclusion

From my research I believe it will be best to use a Recurrent Neural Network, since their inputs can have arbitrary length such that no data will be discarded and they have a long-short-term memory so the previous results can impact the proceeding results. However, this method will increase the processing required to train the network and require more processing to run the analysis, although I believe it is worth the excess processing time to produce a more usable output.

Further analysis of Neural networks

Computational graphs



Computational graphs are a form of directed graph where variables can feed their value into an operation and the operation feed their output into further operations. Here the computational graph computes the sum of two input variables where z(x,y) = x + y
As the computations grow more complex this notation becomes more useful. A compute graph must contain the following components: a compute function which carries out the operation, a list of input nodes and a list of "consumers" which consume the outputs of previous operations. The values fed into the nodes and the output nodes are known as tensors.
http://www.deepideas.net/deep-learning-from-scratch-i-computational-graphs/

More detail on perceptron's

Perceptron's are the constituents of neural networks and consist of weights and biases. These weights can be applied to n-dimensional data to transform them according to some function for example if two data sets when plotted could be differentiated by dividing the data with the line y = x. It could be said that to classify an item of data one could apply the formula:

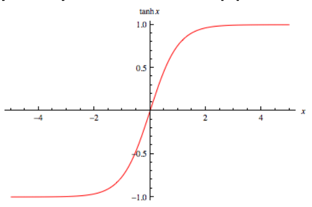
$$w^T x + \ b = 0$$

Where w is the **weight vector** which is then applied to the data points represented by x with the **bias b** applied, the bias will never change after the random initialization and determines if a neurone will usually fire. In this case where the classification is occurring in the line y = x thus the weight w will be (1,1) and the bias 0. Once the weight and bias is applied the data above the line y = x will give positive values while the data below will give negative values.

A perceptron or in other words a classifier can be represented with the notation $\hat{c} : \mathbb{R}^d \ \rightarrow \{1, 2, \dots, C\}$
Where C represents the number of classes. If the weights are $w \ \in \mathbb{R}^d$ and bias $b \ \in \mathbb{R}^d$ the perceptron will be:

$$\hat{c}(x) = \begin{cases} 1, if \ w^T x + n \geq 0 \\ 2, if \ w^T x + b < 0 \end{cases}$$

The resulting value of $w^T x + \ b$ can be arbitrarily high, the higher it is the more likely it is that that point belongs to the corresponding class. In order to convert these values to a probability they must be mapped from $\pm\infty$ to $\pm$ 1 this can be achieved with a sigmoid function such as tanh. It takes any real input and maps it to the required range as shown in the graph.



The function tanh is written $y = \frac{1- e^x}{1+ e^x}$

Alternatively, you can restrict the range of the function to 0 to 1 with the following alteration:

$$y = \frac{1}{1 + \ e^{-x}}$$

N class classification

To classify the data into n classes, multiple operations can be computed on the same vector in parallel, this then introduces a new set of weights and biases which can be applied with the following matrix (for c number of classes)

$$\boldsymbol{output} = \begin{cases} \boldsymbol{\sigma}(\boldsymbol{w}_{T1}\boldsymbol{x} + \boldsymbol{b}_1) \\ \boldsymbol{\sigma}(\boldsymbol{w}_{T2}\boldsymbol{x} + \boldsymbol{b}_2) \\ \dots \\ \boldsymbol{\sigma}(\boldsymbol{w}_{Tc}\boldsymbol{x} + \boldsymbol{b}_c) \end{cases}$$

Where sigma represents the sigmoid function.

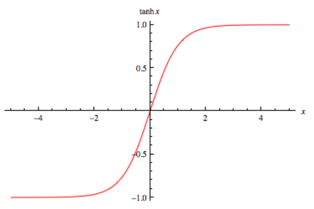
The issue with this is that there are c number outputs to the matrix operation all of range -1 to 1 or 0 to 1 depending on the sigmoid function used. In order for the output to be sent into the next layer of the neural net it must again be normalised, but this time such that the sum of the probabilities is equal to 1. This normalisation can be achieved with the element wise summation shown: $\sigma(a)_i = \frac{e^{ai}}{\sum_{i=1}^{c} e^{ai}}$

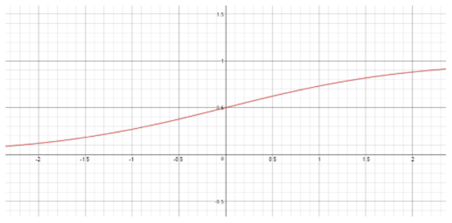
Activation Functions

Activation functions are functions applied to the output of each node in a network to transform their outputs, either to change their range or linearity.

There are two main activation functions used in neural networks mentioned above, both are sigmoid, s shaped curves:

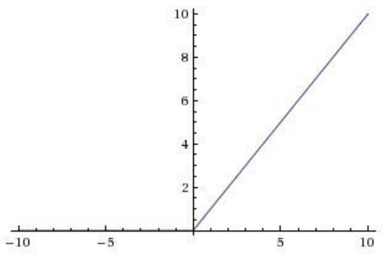
$y = \frac{1 - e^{-x}}{1 + e^{-x}}$ $\qquad$ $y = \frac{1}{1 + e^{-x}}$





They are applied to the output of each node and are used to restrict the domain of each nodes output to a predefined range. Some networks use infinite range functions but these are not as useful in classification programs. The non-linearity of these function is key to their usefulness as the map the vast majority of inputs close to -1 and 1 with respect to the function on the left otherwise known as tanh and 0 to 1 on the right. They also allow for “layer stacking” non-binary outputs unlike some functions which map all values to 0 or 1 corresponding to x values greater than and less than 0.

However, this analogue nature is costly as it makes the network difficult to prune (the removal of unnecessary nodes) as nodes cannot be removed as they almost never output the same value after a sigmoid has been applied. This has brought rise to the ReLu function:



$$y = \max(0, x)$$

This has the advantage of the same output for half of the inputs making the pruning of nodes very easy- simply if the weight is negative. It also has a simple differential but is bounded 0 to infinity blowing up the activation. However due to the functions linear nature it isn’t as well suited to classification. But due to it simple nature makes training larger networks using the ReLu activation function much quicker.

A large convenience of these function is the simplicity of their differential which is used in backpropagation to find the gradient of the weights and biases with respect to a desired output mentioned later. Each differential is as follows:

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

Notation

Each layer of the network has neurons with weights denoted w and biases b in each layer can thus be denoted $w_i \ and \ b_i$ if there are l_k layers, thus each weight and bias could be referred to with w_i^k and b_i^k for the weights and biases at the i^{th} node in the k^{th} layer.
The generalised form of the output of theses nodes could be represented as:
$a_i^k = \sigma(w_i^k * a^{k-1} + b_i^k)$
Here you can see the function is defined recursively using the a^{k-1} this is using the notation:

$$a^k = \sum_{i=1}^{n} a_i^k$$

This represents that a^k is equal to the sum of all the elements in that layer.

This can be extended to a matrix calculation for each layer like so:

$$\sigma\left(\begin{bmatrix} w_{0,0}^l & w_{0,1}^l & w_{0,k} \\ w_{1,0} & w_{1,1} & w_{1,k} \\ w_{j,0} & w_{j,1} & w_{j,k} \end{bmatrix}\begin{bmatrix} a_0^{l-1} \\ a_1^{l-1} \\ a_j^{l-1} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_j \end{bmatrix}\right) \text{jk * j1 = j1}$$

Here the weights of all the neurons can be shown with the k^{th} node in a layer connecting to all n nodes in the next layer, and then the bias is applied to the sum each node in the next layer

Going back to $a_i^k = \sigma(w_i^k * a^{k-1} + b_i^k)$
You can see that the output of any node is equal to the sum of all the outputs from the previous layer multiplied by the neurone's weight and summed with the neurone's bias. The result is passed to the activation function represented with σ.

What is the Loss function?

The loss function is the penalty calculated for incorrect classification, hence the class of the training data must be known beforehand. The higher the output of the function the greater the error in classification. There are two basic loss functions, mean squared, root mean squared and Euclidean distance there are respectively as follows:

MS: $\frac{\sum_1^n (y_t - y)^2}{n}$ RMS: $\sqrt{\frac{\sum_1^n (y_t - y)^2}{n}}$ EC: $\frac{1}{2n}\sqrt{\sum_1^n (y_t - y)^2}$

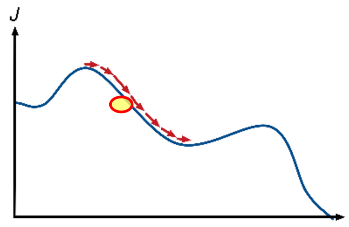
$y_t \ is \ the \ predicted \ class$ $y \ is \ label \ for \ the \ sample$ $n \ is \ the \ number \ of \ samples$

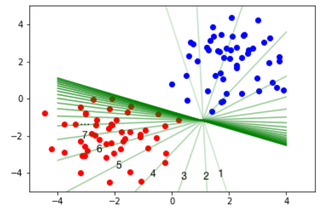
The RMS loss function can also be normalised to account for different scales if the samples are sourced from multiple data sets, this is applicable most of the users will have different mice and mouse drivers. Euclidean distance is halved to make the differential easier to handle.

$$NRMS = \frac{RMS}{y_{max} - y_{min}}$$

Training through gradient decent

Once the structure of the network has been established the weights and bias can be trained such that all of the elements are correctly classified this is achieved through “gradient decent”, where an initial random value for the weights and bias are chosen, they are then iteratively increased or decreased in which ever direction promises the highest correct classification from the training set. If the direction shows promise the interval of weight and bias change is increased until a maximum percentage of chosen classes are correct. If the adjustment overshoots the direction is reversed and interval decreased.





The issue with this method is that it often converges to a local minimum, this would be the point highlighted in the picture on the left, a dip in the graph while there are much lower more optimal possible set of weights and biases possible.

The gradient of the network with respect to network accuracy can be calculated through backpropagation, named this as the error of the value outputted by the network is distributed back through the layers of the network. This however requires the derivative of the “loss function”.

Backpropagation

Backpropagation is the process of calculating the necessary changes to be made to each weight and bias throughout the network in order to give the desired change of the output. This is achieved by propagating the final error back through the network. This final error is given by the imaginatively named error function described earlier.

Because the network must move in the direction of reduced error the negative derivative of the error function is taken. This will give a delta in the output which the weights in the network must change in order to accommodate, the change in the weights should be proportional to this delta to ensure that when the error nears 0 the error is not overshot.

For example if the current output of the network is -0.5 but the desired output is 0.75 the derivative of the error function will be $\frac{d}{dx}\frac{1}{2}(y_t - y)^2 = y_t - y$ $\quad\quad y_t - y = -0.5 - 0.75 = -1.25$

If we then take the negative of this: 1.25 and propagate this change back through the network

Backpropagation uses the chain rule which states that $\frac{dy}{dx} = \frac{dy}{dt} * \frac{dt}{dx}$

From this we can extrapolate that the product of the gradient of all preceding layers will be equal to the gradient of the loss function (the output of the network)

The differential of the mean squared distance loss function for a single sample is shown again below:

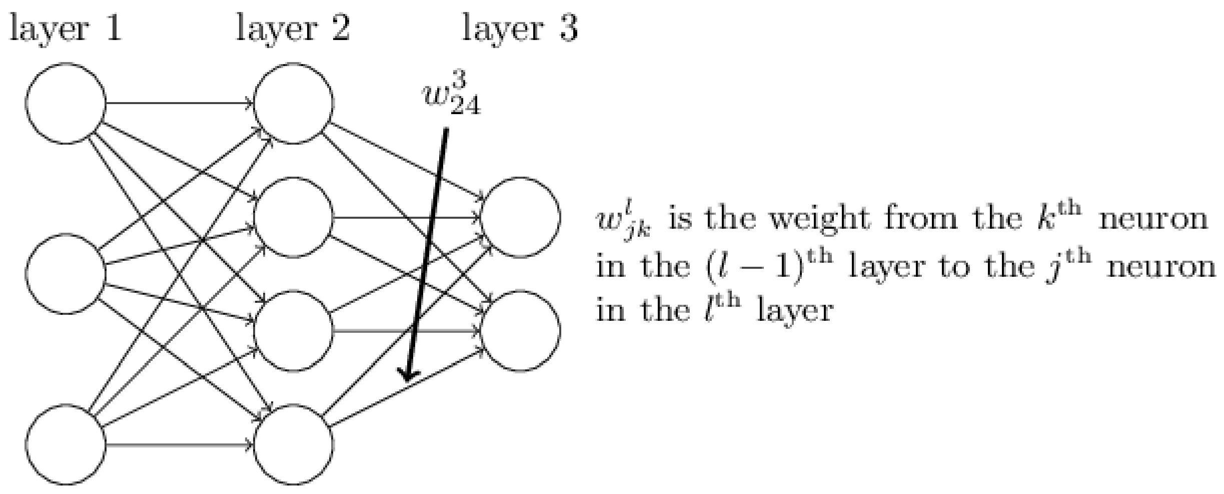
$\frac{d}{dx}\frac{1}{2}(y_t - y)^2 = y_t - y$

Each weight can be changed throughout the network to decrease the error function, this means moving in the direction of decreasing gradient, therefore the negative of the differential of the loss function is used and each weight is changed proportionally to this negative differential. The weights are also changed proportionally to the sum of its inputs as this represents the significance of that neuron in the network.

The following equation is formed when you propagate these desired changes in the network backwards, hence the name backpropagation:

$$\frac{\partial C}{\partial w_{jk}^{l}} = a_k^{l-1}\,\sigma'\left(z_j^{l}\right)\frac{\partial C}{\partial a_j^{l}}$$

The change of the weight of the jk^th neuron in layer l is equal to	The output of the k^{th} neuron in the previous layer multiplied by	The differential of the sigmoid applied to output of the node connected to neurone j	The differential of the total error at node j in the next layer


layer 1
layer 2
layer 3
w^3_{24}
w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

The maths:

In order to update the weights we need to change them in proportion with their gradient with respect to the change error function of the output of the network this this gradient is equal to the change in the error divided by the change in the weight for a given interval. If we denote the error C and the weights w. The gradient of the weight at an index jk in layer l is given by:

$$\frac{\partial C}{\partial w^l_{jk}}$$

We cannot arrive at the correct change in the weight using this formula alone, since we know neither the change in the output of the network for a given interval of the weight nor the change in the weight for a given change in the error. We can however apply the chain rule $\frac{dy}{dx} = \frac{dy}{dt} * \frac{dt}{dx}$ to introduce variables that we do know in the network. For a single forward propagation of the network we know the input for the network and calculate the output, between we also must know the outputs of every layer and simultaneously therefore the outputs of each layer we can denote the outputs of each layer a ,the letter being chosen since it is the value of the activation function applied to the “raw” output of the layer. Prior to the activation function we also know this raw output which can be denoted z. If we use the chain rule twice, we can get the following equation in terms of C, w, a or in terms of C, w, z and finally in terms of all of them: C, w, z, a

$$\frac{\partial C}{\partial w^l_{jk}} = \frac{\partial C}{\partial a^l_k} * \frac{\partial a^l_k}{\partial w^l_{jk}} = \frac{\partial C}{\partial z^l_k} * \frac{\partial z^l_k}{\partial w^l_{jk}} = \frac{\partial C}{\partial a^l_k} * \frac{\partial a^l_k}{\partial z^l_k} * \frac{\partial z^l_k}{\partial w^l_{jk}}$$

we can break down the components of the function above, if we first look at the first operand, $\frac{\partial C}{\partial a^l_k}$ we can calculate the activated output of a node by dividing the sum of all of the raw outputs in the next layer by the weights connecting them to the activated output we’re trying to calculate:

$$a^l_k = \sum_{m=0}^{j} \frac{z^{l+1}_m}{w^{l+1}_{mk}}$$

Therefore, it follows that if we divide ∂C by this new form of ∂a^l_k we can write $\frac{\partial C}{\partial a^l_k}$ as follows:

$$\frac{\partial C}{\partial a^l_k} = \sum_{m=0}^{j} \frac{\partial C}{z^{l+1}_m} w^{l+1}_{mk}$$

The next component I considered is $\frac{\partial z^l_k}{\partial w^l_{jk}}$ this is the change on the raw output divided by the change in the chosen weight. We can re-write this term since the raw output of a node divided by a weight is equal to the output of the connected node in the previous layer thus:

$$\frac{\partial z_k^l}{\partial w_{jk}^l} = a_j^{l-1}$$

Finally we must consider the last component $\frac{\partial a_k^l}{\partial z_k^l}$, the output of a node divided by its raw output, since we can write $a_k^l = \sigma\left(z_k^l\right)$ it follows from rudimentary calculus that the change in output divided by the change in input of the function is equal to the gradient or differential of the function at the input:

$$\frac{\partial a_k^l}{\partial z_k^l} = \sigma'(z_k^l)$$

thus we arrive at the final equation by substituting in our new found equivalencies into the original equation:

$$\frac{\partial C}{\partial w_{jk}^l} = \sum_{m=0}^{j} \frac{\partial C}{z_m^{l+1}} w_{mk}^{l+1} \; * \; \sigma'(z_k^l)\, a_j^{l-1}$$

Error signal

We may also like to know the "error signal" of a neurone, denoted δ. This is the gradient of a node with respect to the error of the network which can be written:

$$\delta_k^l \equiv \frac{\partial C}{z_k^l}$$

However, we can we-write this equation in terms of things we already know:

$$\delta_k^l = \frac{\partial C}{z_k^l} = \frac{\partial C}{\partial a_k^l} * \frac{\partial a_k^l}{\partial z_k^l} = \sum_{m=0}^{j} \left(\frac{\partial C}{z_m^{l+1}} w_{mk}^{l+1}\right) \; * \; \sigma'(z_k^l)$$

We can still do some work here since z and C crop up in both sides of the equation if we reduce the scope like so:

$$\frac{\partial C}{z_k^l} = \sum_{m=0}^{j} \left(\frac{\partial C}{z_m^{l+1}} w_{mk}^{l+1}\right) \; * \; \sigma'\left(z_k^l\right)$$

Here we see that we can from a recursive formula and re-write $\frac{\partial C}{z_m^{l+1}}$ as δ_m^{l+1}

$$\delta_k^l = \sum_{m=0}^{j} \left(\delta_m^{l+1} * w_{mk}^{l+1}\right) \; * \; \sigma'\left(z_k^l\right)$$

We can now use this form of the error signal to greatly optimise training. Since the earlier expression calculating the gradient of the weights with respect to the error of the network requires $j +$ complexity at for each weight of which there are kj many. Resulting in kj^2 complexity for each trained layer.
Instead we can calculate and store the error signal of the output of each layer a d multiply by the activation of the error signal:

$$\frac{\partial C}{\partial w_{jk}^l} = \delta_k^l a_j^{l-1}$$

We now first calculate our error signals for the output nodes of a layer then multiply each of these by the output of each neurone to calculate its gradient. Giving the complexity, $j + jk$! (and no that's not factorial that's an exclamation)
We can now adjust our weights in proportion to their gradient with respect to the error of the network if we know error signal for each layer. With the following pseudocode:

```
For each j in Layer
    For each k in layer
        W[j,k] += ErrorSignal[k] * W[j,k] * LearningRate
```

In order to find the error signal used above we must follow the recursive formula for calculating the error signals:

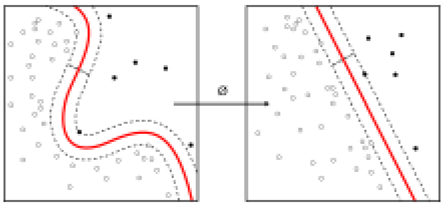
```
ErrorSignal[FinalLayer][0] = ErrorDifferential(Output, Target)
For k in each Layer
    For m in NextLayer
```

```
        Sum +=  ErrorSignal[NextLayer][m] * w[NextLayer][m][k]
ErrorSignal[Layer][k] = Sum * ActivationDifferential(RawOutput[Layer][k])
```

Now we mustn't forget to change those biases!
To adjust the biases we want to know their gradient with respect to the error of the network this can be denoted:

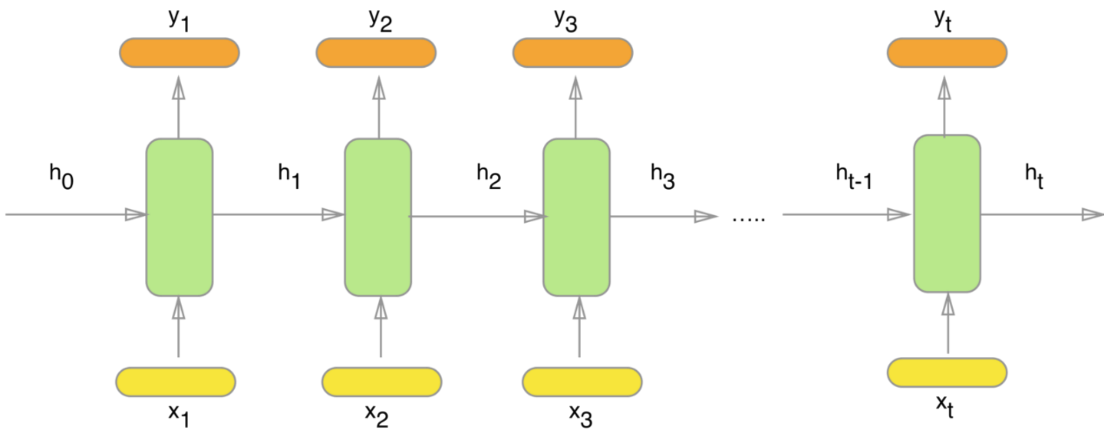
How an RNN differs

RNN's have the ability to exhibit temporal behaviour over time, rather than finding a linear relationship over time like a feed-forward neural network this makes them better suited to recognise handwriting, a data type not dis-similar to mine. An example of this non-linear temporal behaviour can be seen below.



In handwriting this allows for unsegmented font but in my case will aid recognition despite pauses in mouse movement.

An RNN could be viewed as multiple feed forward neural networks, like those described earlier, passing information to each other regarding their inputs, where each network is fed a corresponding section from the input data, like shown bellow:

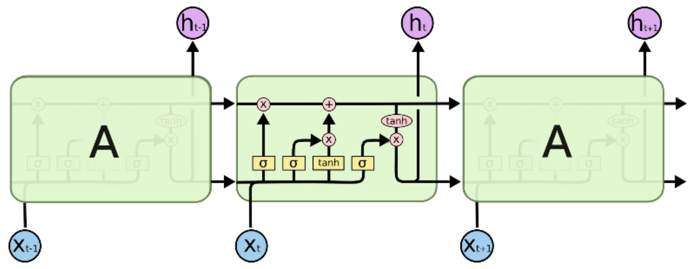


Here the values X_1 through X_t represent t divisions of the input data, each are passed through a neural network which makes a prediction based on its section of the input space,

The issue with the RNN model is the "vanishing gradient problem" due to the same weights being used to calculate Y_t as you progress through the data the network "forgets" about older predictions due to the nature of training with errors tending to extremes the further you move back through the network. Due this LSTM networks have been introduced.

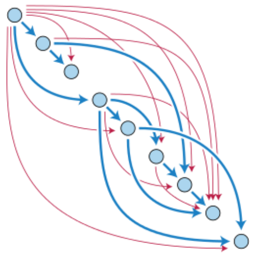
LSTM networks (Long short term memory) solve the forgetfulness of the recurrent model through the addition of a number of layers that interact with each other, this includes the addition of passing additional data to the next layer in the form of the store state, a state vector that is infrequently modified and is passed forward to each following layer allowing LSTM's to learn long term dependencies, this state is passed to the network and is combined with the networks additions for that iteration. The diagram demonstrating the flow of data and processing can be seen below.

(blog, 2015)



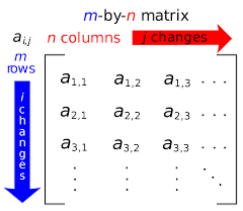
Finite Impulse vs Infinite impulse

There are two main classes of RNN finite and infinite impulse. A both networks can be represented as directed acyclic graphs, a graph plotting n vertex with paths between the vertices where there is no path that cycle from a vertex v through all n many vertices back to vertex v. However, a finite impulse graph can be “unrolled” and replaced with a standard feed forward network while in infinite graph cannot.



2 directed acyclic graphs plotted on the same set of vertices, you can see that there is a path between any two vertices but there is no single cyclical path that encompasses all vertices.

Explanation of matrix multiplication



m-by-n matrix
$a_{i,j}$
n columns
j changes
m rows
i changes
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,1}$ $a_{3,2}$ $a_{3,3}$

$$\begin{bmatrix} 12 & 7 & 21 & 31 & 11 \\ 45 & -2 & 14 & 27 & 19 \\ -3 & 15 & 36 & 71 & 26 \\ 4 & -13 & 55 & 34 & 15 \end{bmatrix}$$

Matrices are essentially n-dimensional vectors with given dimensions such as n x m vector an example of which is shown on the right. Each number in the matrix can be addressed as shown in the diagram with the notation $a_{n,m}$. The matrix can be stored in an array of dimensions n*m whose definition would look like this:

```
Matrix[m,n] as float = (  (1, 2, 3, …, m),
                          (3, 4, 5, …, m + 2),
                          ……………
                          ,(……)        )
```

All mathematical operations can be performed on matrices however they differ from normal operations. The easiest to grasp is addition as, matrix addition is simply element wise addition. The two matrices to be added must be of the same dimensions and each element in the matrix is added to the corresponding element in the other matrix. For instance, given two matrices A and B of equal dimensions you wish to add you must take the x^{th} and the y^{th} of each and add them and place them at index x,y in an output matrix the pseudocode for this is as follows:

```
Matrix1[m, n] as float = RandomMatrix(m,n)
Matrix2[m, n] as float = RandomMatrix(m,n)
MatrixO[m, n] as float
For x = 0 to m – 1
        For y = 0 to n – 1
                MatrixO[x, y] = Matrix1[m, n] + Matrix2[m, n]
```

Multiplication operations are a fair bit more complicated. Take two matrices of size 2x3 and 3x2 and multiply

$$\begin{matrix} 4 & 2 \\ 8 & 5 \\ 7 & 3 \end{matrix} * \begin{matrix} 1 & 8 & 7 \\ 2 & 5 & 9 \end{matrix} = \begin{matrix} 4+4 & 32+10 & 28+18 \\ 8+10 & 64+25 & 56+45 \\ 7+6 & 56+15 & 49+27 \end{matrix} = \begin{matrix} 8 & 42 & 56 \\ 18 & 89 & 101 \\ 13 & 71 & 76 \end{matrix}$$

From the demonstration you can see that a 2 x 3 matrix multiplied by 3 x 2 matrix has a 3 x 3 product, thus I can be said that a matrices of dimensions x, y and y, x have a product of dimension y, y. Thus, matrix multiplication is not distributive since a y, x matrix and a x, y matrix will have a product of dimension x, x. Pseudocode for matrix multiplication would look like so:

```
Matrix1[m, n] as float = RandomMatrix(m,n)
Matrix2[n, m] as float = RandomMatrix(n,m)
MatrixO[n, n] as float
For row1 = 0 to n - 1
      For column2 = 0 to n - 1
            For column1 = 0 to m
                  MatrixO[column2, row1]  += 
Matrix1[column1, Row1] * Matrix2[colum2,column1]
```

GPU Acceleration using OpenCL

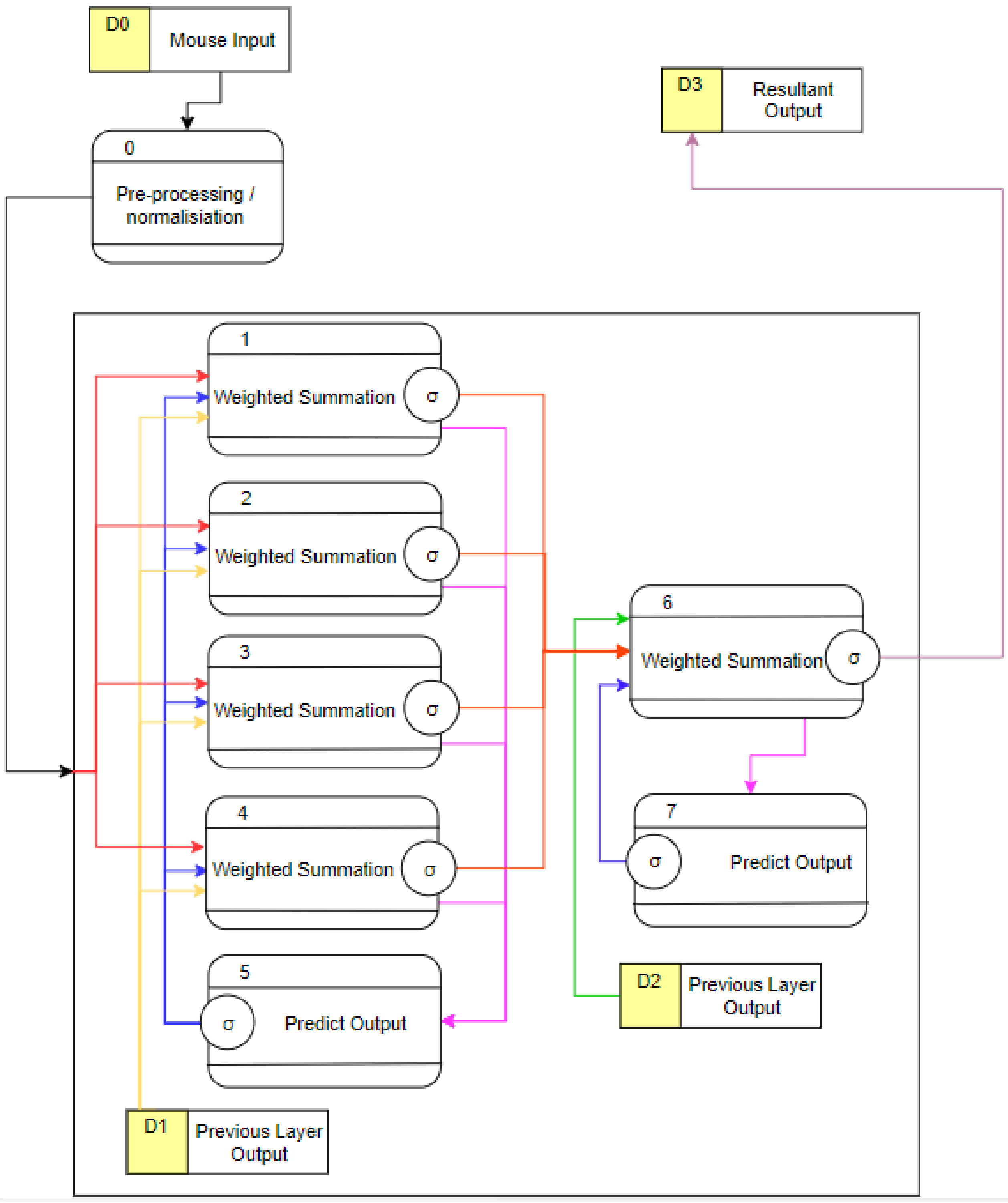
Neural networks require a very large number of floating point operations to be executed especially during training, since a 4 or 5 layer network as I hope to be using with a thousand input nodes will have around 1.2 million floating point operations to perform each time the network is ran, during training the network will have to be ran potentially millions of times requiring Trillions of floating point operations to be made in total. If I ran the training on a an average CPU (2018) with 2 cores and 4 threads @3Ghz such as the Intel(R) Core(TM) i5-7500 CPU training will be ran at 21.47 GFLOPS (Asteroids, 2018), this means I can run 21.47 billion floating point operations per second. In order to improve network performance further I could use GPU acceleration, with my GPU (GTX 780Ti) I have the potential to run training at 5.04 TFLOPS, this means I can execute 5.04 Trillion floating point operations per second! That gives over a 200x performance improvement!

In order to fully utilize the GPU's processing power, I intend to use OpenCL (open compute language) which will allow me to run such matrix operations on any and all available hardware be it CPU or GPU. Officially this makes it a "heterogeneous computing API". It will leverage compute services in parallel – parallel computing. It is an open standard contributed to by Intel, AMD and Nvidia; making it perfect for use in all systems as these 3 companies supply almost all compute hardware for consumers – this means I could leverage the GPU to run the network faster after it has been trained. There are added overheads to consider when using the GPU to perform calculations as the CPU must prepare the data first as the GPU can only compute very specific problems.

OpenCL functions with C++ to multiply matrices by creating a single dimensional array of size m*n (the two dimensions of matrix multiplied together which then get sent through 2 dimensions of parallel threads called "work-items". The ability to parallelise the multiplicative work makes matrix multiplication far quicker when using OpenCL.

In order to use OpenCL, I need the SDK which is only available from the contributors: Intel, AMD and Nvidia each of which have locked down their respective versions such that the code would only work on one of the above platforms. Consequently, I will have to proceed using solely CPU compute when training the model.

DFD for RNN:



This diagram shows how data inputted to the network will be split across all input neurones, during training the neurones adjust their weights outputting the result back into that layer of neurones until their predicted output is correct according to the known class of the input data. The sigmoid functions after each neurone serve to normalise the output between each layer such that the output from each neurone is always between -1 and 1, likewise with the predicted output ensuring nothing outside -1 to 1 is inputted back into the layers. In this example the network s very simple, 4 input nodes, 0 hidden layers and 1 output nodes. In the actual implementation I will have at least 1 hidden layer and will vary the number of neurones in each layer until the network is sufficiently complex to give an accurate output.

Data collection

Mice

Since computer mice will be the source of my data I have decided to conduct some research into how they operate and how data can be collected from them.

The computer mouse

The standard modern computer mouse uses a laser or LED based optical tracker, capturing motion relative to a surface in two dimensions. This implies the surface is smooth, so given a high, resolution mouse bumps in the mouse surface may affect mouse input, since the is no way of knowing how smooth the surface is I will have to assume the surface is smooth throughout the investigation. The resolution of a mouse is defined by its rated dpi, dots per inch, typical mice have a resolution of 400-800 dpi, but more expensive versions feature up to 1600dpi. They also have a specific refresh rate, the number of measurements made per second from 1500 to 6000 per second (by Nyquist's theorem this is more than enough to represent 4-6Hz signals, those found in PD tremors), combining the two is the image processing rate which is the rate at which the data can be processed and sent to the PC, ultimately this is the bottleneck of the resolution of the mouse. All of the above factors affect how detailed the mouse movement is captured, as well as how the movement is scaled; consequently, I will have to normalise the mouse data from each source to account for the resolution and sampling rates of each otherwise errors could occur when the data is analysed
(Wikipedia, 2018)
(Carmack, 2016)

Tracking

In summary, most tracking software uses web-based applications but they all simply record co-ordinates of the mouse position and log them to a file. MouseTracker.org offered "normalised" or "raw" time, normalised being a timestamp that has been converted to an easy format, either a decimal of known length since the start of the program or a known format such as minutes, seconds, milliseconds … This allows for easy comparison between values and between various sources of data. While raw values could have a different scale or could be in an erroneous format for analysis. MouseTracker.org also "generates mean trajectories of conditions and computes indices of spatial attraction/curvature and complexity", plainly this means it can smooth noisy mouse data by simplifying multiple co-ordinates into a line or curve. But this intrinsically reduces the resolution of the data, which could help the analysis method distinguish between Parkinsonian and normal mouse movement. The majority of these trackers store the output in a CSV format, suitable for viewing in a spreadsheet program such as excel.

CSV Storage

CSV files store the data in a 2-dimensional array with the columns of data separated by commas and the rows by carriage returns. This has the implication of storing the decimal values as strings of text. Consequently, each digit would require 1 ascii character of storage (8 bits) plus would have to be converted from a string to decimal when a calculation is to be made with that piece of data and vice versa when the data is stored. I believe this would add an unnecessary overhead to the program having to convert back and forth between data formats as well as require far more storage. This is a real issue since I intent to be collecting a lot of data and don't want to introduce unnecessary processing overhead since the data collection will be running in the background of user's machines so mustn't consume a lot of resources be them memory, storage or processing. Consequently, I will be investigating raw storage techniques later.
Src: (Freeman, 2018)
The above source also led me to finding the following sources (Notebaert, 2017) (L.Abrahamsea, 2016) Which were both able to draw conclusions from the collected mouse data.

Microsoft – Win 32 C++ Mouse movement capture

Win 32 (32 bit windows kernel API) posts a WM_MOUSEMOVE message whenever the cursor is moved, the message normally goes straight to the active window (the program currently being used); but this can be changed.

The WM_MOUSEMOVE message contains the X and Y mouse Co-ordinates 16 bits each.
I realised this may implement a restriction of x and y screen resolutions being less than 66535 pixels across and may be adversely affected by multiple monitor displays.
I did some research which lead to the solution found here (Codeka, 2009), Accordingly there is an alternative to WM_MOUSEMOVE, WM_INPUT which captures raw mouse data and is usually much higher resolution, the X and Y co-ordinates are stored as longs, which are 4 byte each or 24 bits long and have a range of -2,147,483,648 to 2,147,483,647 giving far more resolution than the processed 16bit data, I believe this will help recognise the symptoms since early signs are supposedly very subtle. WM_MOUSEMOVE and WM_INPUT are both handles, abstract references to a resource, often memory or an open file. The handle serves to hide the actual memory address from the user. Once I have retrieved the WM message I can lookup the handle it stores and use that handle to access the corresponding resource, in the case of WM_INPUT this will be a data buffer of co-ordinates.

Reading Operation

In order to access the correct input device, the application must first collect information about the connected input devices, this is done by GetRawInputDeviceList function which retrieves an array of devices in a RAWINPUTDEVICELIST structure.
Using this array of RAWINPUTDEVICES you can then retrieve information about the given device with the GetRawInputDeviceInfoA function and request basic device info. This returns a device info structure which gives the sample rate per second! This will be useful later when I come to normalise the mouse data. However, it does not give the sensitivity of the mouse (DPI) which is arguably more important.

To get mouse movement data I must first gain access to the WM_INPUT message which is restricted to applications that have the correct TLC (top level collection for the mouse this is within usage page: 0x01, and has an ID: 0x02). I will set `dwFlags = RIDEV_INPUTSINK` to allow me to collect mouse data regardless of the window currently being used. The TLC is achieved by registering access to the corresponding raw input device (mice for me) this is done with the `RegisterRawInputDevices` this must be done with refence to a running process, for me that will be the process itself. I can then call get message and capture messages posted to a window. For which I need to create a window to reference this is achieved by defining a windows class with all the attributes that define the type of window you want. If the message name equals WM_INPUT the lParam parameter will contain the X and Y coordinates as the low and high word the top and bottom 32bits.

(Microsoft, 2018)
(Microsoft, 2018)

Internet Data Transfer handling

How is Data transferred?

Computers communicate with the TCP/IP protocol, like a recipe of what computers must do to send data over the internet. It defines what data to transmit, when to transmit that data and how to transmit that data, if the rules are not followed a connection will not be established. Each home network is assigned a unique IP address provided by their ISP, their internet service provider, this address can be used communicate with a given device on the internet. In a home network all the computers and other devices are connected to a local area network (LAN) to a main network router. The router then manages a connection the clients ISP, from here the LAN is a member of a wide area network (WAN) controlled by the client's ISP. When you access a website via its URL the IP address of that server is found through the use of a dynamic service provider (DNS) who's IP address is known, the client requests the IP address corresponding the URL. Once the IP address has been returned the client can exchange information with the server again using the TCP/IP protocol. (Hope, 2017)

WAN data transfer

One way of implementing my solution will be having a computer at home or in college collecting data from PD sufferers, this implementation will be connecting two client PC's without dedicated URL's. In other words, two peers known as a Peer to Peer (P2P). In a peer to peer network all computers have equal privileges and can act as both clients and servers. However, they are notoriously difficult since the WAN IP address of home users changes regularly since there are insufficient IPV4 (Internet Protocol Version 4) addresses to give each router a permanent unique IP address. This is due to IPV4 consisting only of 4 bytes of data, giving only 4,294,967,296 addresses globally, this is not enough for the 24.3 Billion IOT (internet of things) devices globally so ISP's share addresses reallocating them each time a device needs to connect to the internet. This means I would have to use a dynamic DNS (DDNS) service such as "NoIP" to assign myself a URL that can be updated with my current IP address whenever it changes, if I went down this route I would have to implement the solution as client-server network with me having to run a computer at home 24.7 as a server to receive mouse data I'd also have to reconfigure my router at home to connect to a DDNS server and forward a port to my server, in summary this would be very time consuming and costly as well as security risk to my home network as there would be no firewall when connecting to that port on the router. (Statista, 2018) (Wikipedia, 2018)

WinSock

My investigation calls for files to be transferred from a client PC to me in some way. This could take form as a "peer to peer" transfer this is possible with WinSock, the main tool shown on all C++ based networking tutorials I have seen, a programming interface that handles internet applications from windows following the Windows open system architecture (WOSA) which defines a standard service provider interface (SPI) configurable through the application programming interface (API), especially meaning WinSock handles the services that are configured through the WinSock API. It deals primarily with TCP/IP connections but can handle several protocols.

One Drive

Another potential solution is to use a web server to store the files, this server will have a URL already so its IP address is easily accessible through the use of a DNS, the files can then be temporarily stored that server and can be offloaded every afternoon by me to prevent the server from running out of storage space. A potential host would be OneDrive which I have through my college account. I could then use the REST API from Microsoft to control the upload of files to my OneDrive account. This method would be free and alleviate the need for me setting up a DDNS server and allow easy storage. (Microsoft, 2018)

AWS

Alternatively, Amazon offer a free service called Amazon Web Service (AWS) offering 25GB of database storage. They also have a C++ dedicated SDK with easy client-side encryption.
AWS is a 24.7 service offering virtual computers to perform a variety of tasks depending on the clients subscription, the free tier offers 25GB of DynamoDB storage through NOSQL database service. NOSQL allows

for the storage of almost any datatype rather than the usual tabular form of SQL based relation databases such as Column, Document, Key-Value or Graphical storage types. The DynamoDB service also allows for 200M requests to be made each month which is more than sufficient for my use case. AWS also offers 750 hours of Amazon EC2 resizable compute capacity in the amazon cloud, offering Linux, RHEL, or SLES t2.micro instance usage. 1GB of cloud based analytics. 750 hours of Amazon RDS, data base management time. 5GB of standard storage. 250 hours of Amazon SageMaker ML model development time. And 1 million AWS Lambda requests to be made, running basic computation based on events. However after reading the small print I have found that AWS free tier is only offered with a 12 month free trial
(Amazon, 2018)

Handling shutdowns

An issue I will have to resolve will be handling system shutdowns, as I will have to insure that all the collected data is uploaded to the server otherwise it will be lost. My initial thoughts on the issue are to either stream the data to the server constantly or to upload all the data collected in X sample intervals and to catch shutdowns and upload the data before the program is halted.

Stream

Form some early research I believe that streaming the data to the server will be rather processor intensive as many thousand potential sample uploads will be initialised.

Buffered upload

The majority of sources seem to be in favour of a buffered upload, in which you wait for a sufficient amount of data before uploading it to a server. Since this is far less processor intensive than the stream of many small packets but will not result in a significant data loss due to restarts provided the buffer size is minimised.

Conclusion

To conclude I think that using AWS will be the most effective solution due the large suite of options available to me and the ease of implementation through their well-documented, open source, C++ SDK. Allowing object-oriented management of AWS services rather than One Drives REST API which is deprecated code and designed for C so offers no object-based facilities. WinSock would have been the ideal solution given that I had a server with a static WAN IP, since I don't it will be too time consuming to configure.

The Fourier transform

I wish to apply a Fourier transform to improve the accuracy of the neural network as from my earlier research (Adams, November 30, 2017) it showed a drastic improvement in the accuracy of ML algorithms applied afterwards.

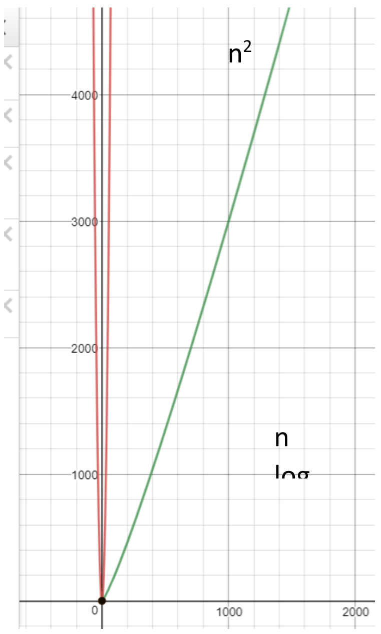
Fourier transforms function by decomposing an amplitude time signal into an amplitude frequency interpretation. An example of this would be a 50Hz sine wave applied to a Fourier transform would output a vertical line where X = 50 of equal magnitude to the RMS magnitude of the input sine wave.
(Wikipedia, 2018)

The Fourier transform in principle works by mapping the samples onto a circle incrementally and measuring the centre of mass of the resulting deformed circle within each interval, broken down for each element in the array the sum of all the elements is taken as a real and complex component of the magnitude of that sample If it was mapped onto a polar grid (a grid where points are represented by their magnitude and their argument, the angle above the initial line (positive x axis)) with incrementing arguments such that the argument of the last index is 2pi. The resulting real and complex sum at that index represents the magnitude of the that frequency at that index. Usually Fourier transforms are only applied to 2-dimensional data, amplitude time, however my input data has 3 dimensions, X, Y and time. This has the effect of needing to map the data onto a sphere with a pair of complex co-ordinates. Algorithms generally use the real component of these co-ordinates so the output will be an N element long 2-dimensional array.

There many algorithms to transform data into a frequency amplitude domain. The standard algorithm has n^2 complexity, this is not ideal as it would take exponentially long times to transform the large data sets that I intend to input into the network. An example in pseudocode of a basic n^2 complexity algorithm is shown below:

```
def discrete_fourier(x):
  N=len(x)
  fft = []
  for k in range(0,N):
    accum_r = 0
    accum_i = 0
    for n in range(0,N):
      tmp = -2 * math.pi * k * n / N
      accum_r = accum_r + x[n] * math.cos(tmp)
      accum_i = accum_i + x[n] * math.sin(tmp)
    fft.append(complex(accum_r, accum_i))
  return fft
```

(Own knowledge)

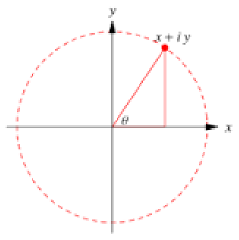


The main alternative algorithm is the fast Fourier transform (FFT) an n * log(n) complexity algorithm, the most used variant is the Cooley Tukey transform:

```
void fft2 (complex<double>* X, int N) {
  if(N < 2) {
    // bottom of recursion.
    // Do nothing here, because already X[0] = x[0]
  } else {
    separate(X,N);     // all evens to lower half, all odds to upper half
    fft2(X,     N/2);  // recurse even items
    fft2(X+N/2, N/2);  // recurse odd  items
    // combine results of two half recursions
    for(int k=0; k<N/2; k++) {
      complex<double> e = X[k    ];  // even
      complex<double> o = X[k+N/2];  // odd
                 // w is the "twiddle-factor"
      complex<double> w = exp( complex<double>(0,-2.*M_PI*k/N) );
      X[k    ] = e + w * o;
      X[k+N/2] = e - w * o;
    }
  }
}
```
(Wikipedia, 2018)

This functions by splitting up the standard Fourier transform into half’s (all odd index’s and all even index’s) and recursively carries out the transform on each half of the tree until the array has shrunk to 1 index, within each recursion the sum of each half of the tree is taken as a complex point mapped as if it was on a circle with angle 2*Pi*k/N and magnitude equal to the previous recursions. This technique is often referred to as the divide and conquer strategy.

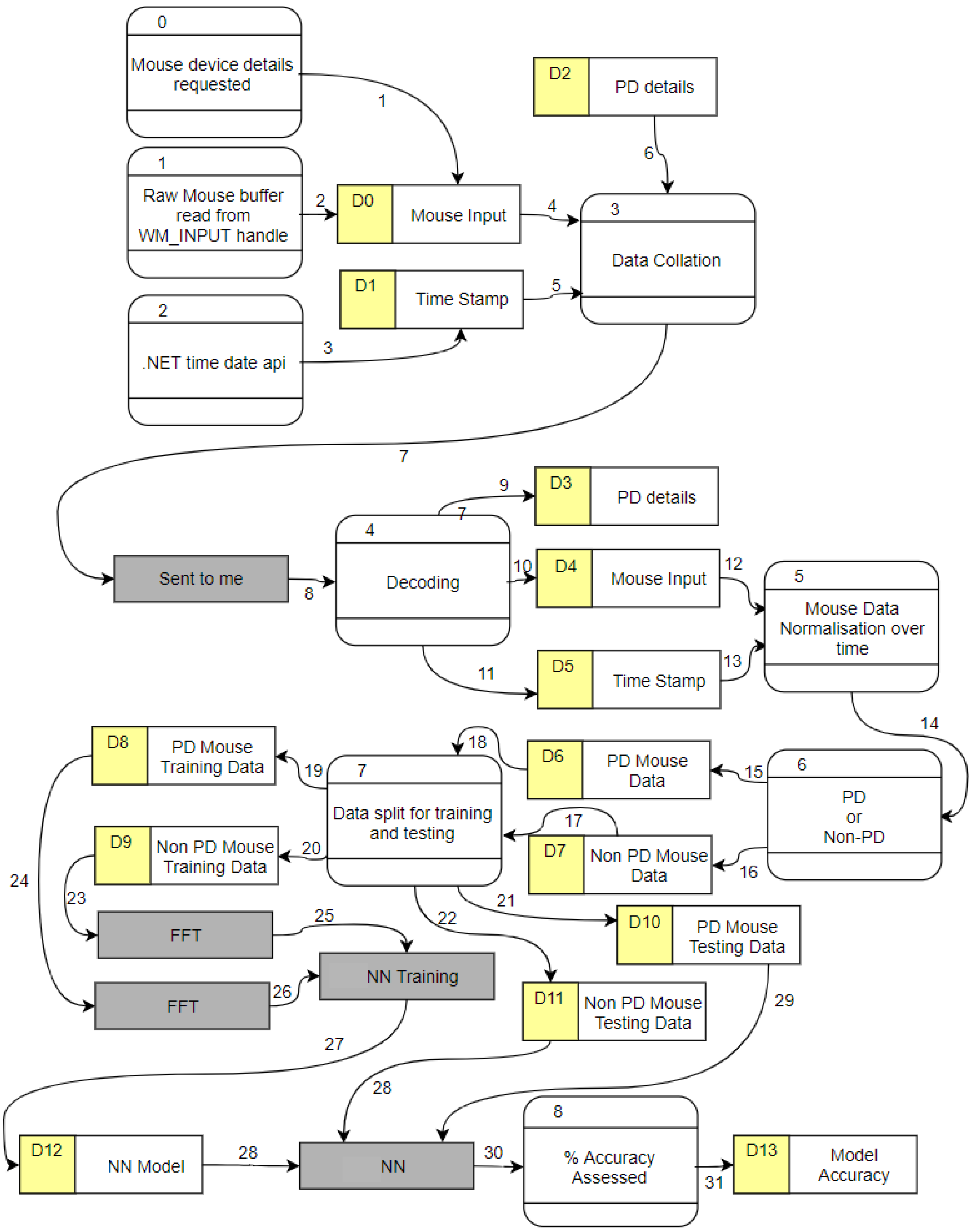
The function takes two parameters, an array of complex numbers and the number of complex numbers in the array. Complex numbers are numbers that have both a complex and real component. The complex component is any multiple of $i = \sqrt{-1}$. The Fourier transform uses the two parts of a complex number to act as coordinates over time, complex numbers are usually represented on an argand diagram shown.



In my scenario the complex part of each element in the array will be the magnitude of the mouse coordinate, the issue occurs due to the fact the Fourier transform requires even sample intervals. Thus I must interpolate between the samples to gain a common sampling interval.

Solution Summary

Level 1 Data Flow Diagram



Brief description:

Frist mouse details are collected and stored in the Mouse Input data store then the mouse co-ordinates are streamed into the mouse input data store. Meanwhile the corresponding time stamps for each set of co-ordinates is streamed into the time stamp data store. Once a data set has been collected and stored the data is collected together into the mouse location over time and stored in a file, the details of the user (whether they have PD or not) are then stored as a header in the file (this data will be collected when the program first runs through a simple form). The resulting file containing mouse data over time and the persons details are then 'Sent to me' as a separate process; I have delved previously into how this will be achieved. I then decode the data that has bee sent to me based on the mouse details, collected data and personal details. This can then be normalised such that the rate of change of mouse movement is set to the same scale across all samples such that the RNN can compare across all data samples. This normalised recording is then sent to the corresponding class of data store PD or Non-PD such that I have an array of recordings from both, people with and without Parkinson's; I can later use that data to calculate the accuracy of my RNN. Next the data will be split in the ratio 90:10, for training vs testing according to (Chamikara, 2014). The training data will then have an FFT applied to it since this helped improve results according to some sources and then will be passed to an RNN training procedure. The training model will then loop until the model is correctly classifying the training data at least 75% of the time, although I hope to achieve better results than that. Once the model has been trained it will be tested on the remaining 10% of the data, the accuracy of the classification will then be evaluated and outputted for storage. If the model is not sufficiently accurate I will adjust the network configuration until optimal results are achieved.

Solution requirements

1. A system that can decide with a certainty of at least 75% whether a person is believed to have PD or not
2. Collect highest possible detail mouse data
    a. Is capable of clearly showing tremors
        i. Mouse data is collected at the maximum resolution of the mouse which is stored
    b. Sampled at a frequency of at least 6*2Hz
        i. Due to tremors occurring at 4-6Hz multiplied by 2 due to Nyquist's theorem
        ii. Mouse refresh rate detected
    c. Recognise new mouse devices when they are connected
        i. Record available mouse information
    d. Does not crash when all input devices are disconnected
3. Data collected is stored in a clear and known format that easy to view while taking minimal space
    a. Data can be exported to Excel for visualization
        i. Exported data is split into three columns, mouse X,Y coordinates and time
        ii. All logs have headers that are easy to understand
    b. Data is stored in binary for minimal file size
        i. A predefined data length is established that allows for storage at maximum resolution across all input devices
    c. Data can be continually stored with a stream
        i. Stream does not pause
        ii. Stream does not require large memory buffers >500MB
4. Data is transmitted to a web server for storage
    a. Data being transmitted is encrypted
        i. Can be quickly encrypted and decrypted efficiently without utilizing to many system resources
    b. No data is lost as it is stored
    c. Data can be accessed at all times
    d. Restarts do not cause the loss of any data, i.e. the data must be streamed to the sever
5. Create a model of Parkinson like mouse movement
    a. Model is scaled the same as mouse input
    b. Model uses a smoothed "healthy" input
    c. Has a 4-6Hz sine wave applied to it
    d. Contains noise with sporadic peaks
6. FFT used to Pre-process
    a. Uses the Cooley Tukey recursive algorithm
    b. Can be ran in n*log(n) time
7. Multi-Layer recurrent neural net training
    a. Model can be trained with minimal compute power, taking a matter of days of compute time on a i7 machine
8. Multi-layer neural net to process user mouse data
    a. Is more than 75% accurate
    b. Displays output clearly if requested
    c. Does not crash given any extreme mouse data
9. The entire solution can be ran on a client machine with minimal impact
    a. Solution uses no more than 10% of the CPU (on an 4 threaded processor)
    b. Solution uses less than 500MB of RAM
    c. Causes no mouse "stutters" or unresponsiveness

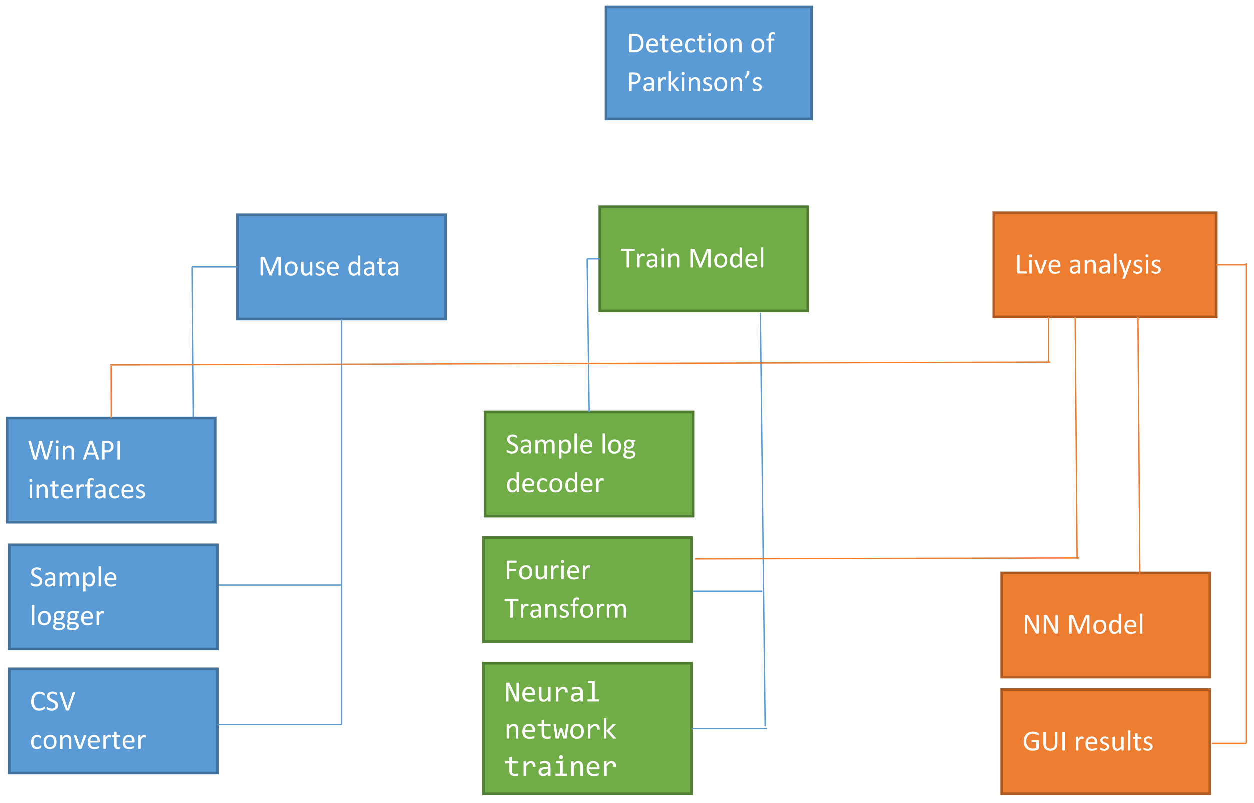
Requirements blurb

ID	description
1.	This is the main requirement for my project that it can correctly classify between parkinsonian and non-parkinsonian mouse data. This will be verified by testing the final trained model on all available mouse data at various severity's of Parkinson's simulated and measuring the percentage of correctly classified samples. If it falls above 75% this requirement will have been met. The 75% value was found from my research as the current accuracy of tests.
2ai.	The system collects mouse data at the highest resolution possible for the device, usually higher then the resolution windows converts to, the higher the resolution the solution is able to collect the more data it will have to make predictions with.
2bi.	Nyquist's theorem states that: an analogue signal waveform may be uniquely reconstructed, without error, from samples taken at equal time intervals. The sampling rate must be equal to, or greater than, twice the highest frequency component in the analogue signal. Consequently, although the mouse movement may have compents of frequency higher than 6hz, since the solution is only detecting components in the range 4-6Hz the maximum required sample rate to reproduce these frequencies would be double the max frequency I wish to reproduce therefore the solution must be able to sample mouse data at 12Hz.
2bii.	The mouse refresh rate is determined using windows API tools, this can be done algorithmically or by requesting the value from the mouse device.
2ci.	Windows can handle a multitude of mice being connected simultaneously which can all be used equally to control the windows environment, the best example of this is laptop users with mice as the trackpad will remain enabled while the mouse is used, they may even alternate between trackpad and mouse use.
2d.	The method employed in the solution can correctly handle the disconnection of all mouse devices and their subsequent re-connection, as this may occur for desktops with wireless mice or for what ever reason the usb is removed. This is less of an issue for laptop users but in the eventuality the solution should not miss any opportunity to log data due to a crash of this cause.
3ai/ ii/iii	The mouse data export should be easy to understand as the user should be able to clearly see what data is being collected about them
3b.	Binary storage implies that the raw binary values for each sample are written to a file, this file will not be readable unless with a hex editor as no values represent plain text, such as ascii. This storage strategy is crucial as it minimised the size of the data stored on the users device especially since a large amount of data is intended to be collected, it will also increase the speed of file writing as less data will have to be written to the secondary storage medium.
3ci.	Data is continuously streamed to the file such that no mouse data is missed while the data is being stored. This strategy also spreads the load over time on the secondary storage device.
3cii.	Application uses the minimum amount of memory possible as the application should have the minimum impact on system resources as intended to be a background task.
4ai.	Data sent over a network connection is encrypted, although the data is not extremely sensitive the program should comply with the data protection act to minimise the chances of a data breach
4b.	Data cannot be lost as it is sent over a network as this could cause anomalies that the neural network may train to spot rather than differentiating between actual parkinsonian characteristics.
4c.	The server storage must be designed such that data can be retrieved at all times, even when clients are connected and uploading files, this requirement ensures that data sent to the server can actually be read from the server.
4d.	Any and all data sent to the stream must be error checked and should be streamed for minimal data losses during upload, in the event of a restart.
5a./b /c/ d	Compare normal data and that which has had Parkinson's simulated, the two sets should look similar with smooth motion over the same scale but the parkinsonian data should show a 4-6hz sinusoidal nature as well as occasional cliff like features.
6a.	Operation of the Cooley Tukey Fourier transform can be demonstrated by stepping through each operation
6b.	It can be shown that the number of iterations for a given input size are consistent with the theoretical n log n timings

7a.	The training component of the project should be able to execute without severe bottle-necks (when the given component is the primary cause for the performance limitation) when using an i7 which I take to be ab approximately 3Ghz or more processor with 4 cores and hyperthreading.
8a.	The neural network library is globally 75% accurate on suitable testing examples
8b.	During training the progress of the training can be clearly seen over time showing the classification of various test classes
8c.	The network can handle a variety of data which may cause incorrect classifications but not for the network to crash at any point.
9a/ b	The client that runs on the end users machine does not use too much of the available system resources, at most 10% of the CPU’s processing performance and 500MB of system memory as this leaves the user with enough for every day tasks without noticing much of a difference.
9c.	The client does not cause any unresponsiveness due to requesting mouse data or network processing as this could hinder the users experience and lead to frustration especially if the machine is used for work.

Design

The project will consist of 3 main parts made to operate in turn during different stages of the investigation. The first is the Mouse Client, this is a small client that will use minimum system resources to collect and upload mouse data from a population of computer users to be sampled to a file storage server using authenticated and encrypted network transfer.
The second the model trainer, this will take the collection of data from the population and superpose the desired Parkinsonian characteristics on a subset of the imported data (half of the data will have the simulation applied as the network must not demonstrate a bias from training) and use the two data sets generated to train a neural network to recognise each class with an accuracy greater than 75%, the network will be trained for as long as possible until the outputs have converged for long as possible.
The final section is the live detector, this will apply the network trained previously in a small client to run in the system tray which will log the mouse data through the same methods as the mouse client, and once sufficient data has been collected classify the data using the given network. The results will then be displayed graphically when the user has decided to view them.



Mouse Data

WinAPI Interfaces

Windows API interface creating a hidden window registering corresponding window classes and raw input devices, then monitors messages posted to the hidden window, decodes them and filters for raw mouse data – returning the change in X/Y co-ordinates since the previous message. Also retrieves time stamp according to the systems clock, returning the time delta between each sample.

Sample logger

Writes mouse samples and time stamps to a binary file with minimum processor utilization. As well as managing a FTP file upload stream to a server defined in a decoded config file

CSV converter

Takes binary file(s) from the sample logger and convert(s) them to comma delimited strings as a CSV file for viewing in Excel

Train Model

Sample log decoder

Takes a binary file from the sample logger and reads all values into arrays, for X coordinates, Y coordinates, the time in microseconds starting from 0 and mouse sample rate.

Fourier Transform

Recursively divides the samples (with even intervals) into odd and even indexes computing the magnitude of the frequency at each recursion out from the terminal call in the stack. Applied to each vector inputted to the trained neural network

Neural Network Training using backpropagation

Train weights and biases of a neural network for application in the feed forward neural network during live analysis, achieved through backpropagation using labelled classes from a large collection of logs.
This will rely on an extensive matrix mathematical library I intend to develop, which can also utilise a GPU for accelerated network training. All neural network and matrix libraries will be fully parametrised so different configurations can be tested.

Live analysis

WinAPI

Will act much the same before but won't call any sample logger processes

Fourier Transform

An identical Fourier transform as was used for training

NN Model

Model applied linearly to as much of the data sampled as possible, while data is being collected compounding to a given prediction

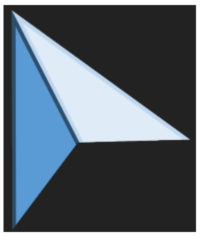
GUI results

GUI to be ran from an icon in the system tray as to be subtle, displaying options to view the current results graphically over time, and a final prediction. With options to start/stop collection and exit the application.

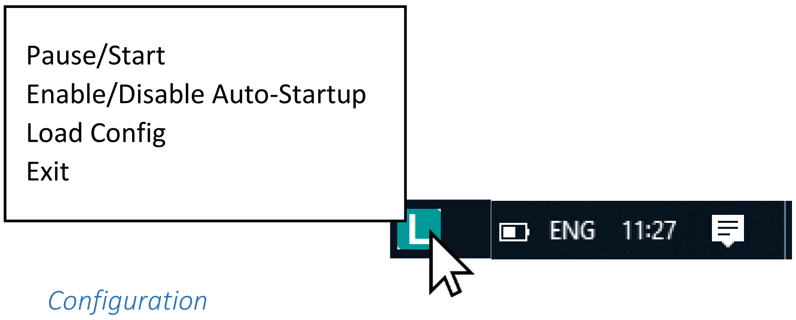
Mouse data Collection

Interface

I intend for users interface with the mouse client through the use of a system tray icon with a context menu, this is a popular method for background windows apps to interface with users as it allows the user to quickly access the program without using valuable screen space. For this I need an icon to represent the program so have created the following image – I kept in mind that any interfaces must not demonstrate that the user is testing for Parkinson's so chose a mouse-like icon as shown.



When the user clicks on the icon a menu will be displayed with the utility options for the program, these will consist of being able to pause/start collection allowing the user to decide what data is sent. There will also be options to enable/ disable auto starting the mouse client data collection program with windows. Next there will be an option to exit the program without removing it from auto-start. Finally an item to load a configuration from a file. The configuration will be a text file with a structure described later to update the configuration of the program. I will also add an option to hide the menu so it is clear how to do so
The menu items will manifest in the following order:



Configuration

The configuration file will be used to store information concerning desired server to upload logs to during data collection. It will contain the URL of the target server, and any authentication details of the FPT server chosen to upload the files to. The configuration file will be stored in the same directory as the application and can easily be replaced in order to change the target server or update security settings in the event of a breach adding desirable flexibility to the application. On application start up the program will Import the configuration file retrieving the URL, username and password which are read as strings after the following keywords "URL: ", "Password: ", "Username: ", until the next carriage return or the end of file is reached. If there is any error when reading the username or password strings, authentication to the server will be disabled. However, by default the .Net libraries FTP server API provides automatic SSL encryption, given that the server supports it, reducing the chance of data being comprised, I believe this to for fill my requirement sufficiently since the mouse data does not explicitly contain any personal information and any that could be gained would not warrant the loss in performance due to complex encryption processing. On reading the configuration if the URL is invalid a flag will be set to disable uploading all together, however local logging to a file will continue. When the user clicks on the "Load Config" label they will be prompted with a windows file explorer window to locate the path of a new configuration file; the new upload details will be imported from the file like they are on start-up and the URL, username and password will be overwritten.

Auto-start

Auto starting the program will by default be enabled, allowing the program to automatically start up with windows. This will function by locating the windows register at the location below: `Software\Microsoft\Windows\CurrentVersion\Run` and then writing the string representing the path of the mouse client executable, when the application is first run. Since windows checks this location for Auto-Start-up items the client will then be launched on each start up. Consequently by default the menu will display the label to disable Auto-start. If disable is chosen the registry key will be deleted erasing the mouse client from the start up menu. If it's enabled the same process is carried out as at the start of the program creating the key in the register. At every stage of deletion and creation the status of the register will be checked to ensure no errors have occurred, since the register, if corrupted could corrupt the users operating system. If an error occurs the user should be notified with a message box describing the error.

Pause/Start

The mouse client will also have the ability to pause all logging, this will be controlled by a flag that will cause the message loop to skip adding samples to the buffer and prevent any local logging or FTP uploads. The message loop will still run to maintain functionality as the user interface of the program relies on the message loop functioning, I believe this to be ok as the utilization of resources will still be greatly reduced and the user will always be able to exit the program fully if they wish to conserve all resources possible. By default the flag will be set to enable logging so the label will display the option to "Pause"; on clicking the flag will be switched and the label changed to "Resume", in addition the task icon name will switch to logging paused to remind the user if they pass over the icon. If the user decides to resume the label will be switched back to "Pause", the flag will be switched and task icon name switched back to mouse client.

Exit

The exit label will result in a halt message being sent to the application, on reception of the halt message all data structures and allocated memory will be freed as well as de-registering any window classes and mouse access. In addition to this the log files must be safely closed and any file uploads to the FTP server must be checked and ensured successful. Upon verifying that the above has been completed the program will halt and the application will terminate. All registry edits will remain unchanged however – maintaining auto start up if it has been enabled. On exit the program will also be removed from the users system tray to prevent a "ghost" icon.

Menu Pseudocode

```
OnPause(){
      LogFlag = false
      SetLabel("Resume")
      SetIconName("Logging Paused")
}
OnResume(){
      LogFlag = true
      SetLabel("Pause")
      SetIconName("Mouse Client")
}
SetAutoStart(){
      String Path GetEXEPath()
      SetStartupRegister(Path)
}
RemoveAutoStart(){
      DeleteStartUpRegister()
}
LoadConfiguration(){
      File = GetFileUsingFileExplorer()
      File.open()
      Index = File.SearchFor("Password: ")
      Password = ReadStringFrom(Index)
      Index = File.SearchFor("Username: ")
      Username = ReadStringFrom(Index)
      Index = File.SearchFor("URL: ")
      URL = ReadStringFrom(Index)
      If(Password or Username = NULL){
            Authentication = false
      }
      If(URL = NULL){
            Upload = false
      }
}
```

The simple pseudocode above demonstrates the core functionality the system tray menu will rely on when each menu item is selected. It also illustrates the required variables to operate the tray message loop. OnPause Is called when the pause button is clicked, it sets the log flag to false to disable logging; next it changes the label of the button the user clicked (Pause) to "Resume" so the user knows the function of the label has switched, finally it switches the name of the icon in the system tray to "Logging Paused" such that when the user hovers over the system tray icon "Logging Paused" appears by the mouse cursor.

Data collection

The mouse clients main purpose is to log X and Y co-ordinates over time to a file which can be sent to a server for storage, this data will then be processed and used to train the neural network. Since the processing phase requires the application of an FFT all samples must be evenly spaced, processes used to achieve this are discussed later, thus we must store a time stamp, since there are numerous ways of storing the 3 data elements I have documented how I intend to store them bellow:

I will be using the windows 32bit API to collect mouse data through the use of the WM_INPUT message, which holds the change in X and Y coordinates since the last message was posted, represented as 32bit signed integers, to reduce the number of operations in preparation to store these values I will be storing the raw delta in the sample log, consequently the first 64bits in each sample in the log will represent the X and Y coordinates respectively, each using 32bits. To record the time I will use the C++ standard libraries Chrono header, it's methods can be used to get the current time since the epoch (January 1st 1970) in microseconds, this gives me a high enough degree of accuracy to store mouse movement of a frequency up to 20kHz (assuming a given mouse had a sample rate of 40kHz). However microsecond precision results in the time stamp being a 64-bit unsigned integer. In order to make the recorded data easier to interpret, I will store the time delta between samples. Otherwise if the user exports the data, as I stated in my requirements, it will be difficult to differentiate between each of the very long integers. In order to ensure that the time stamp is, under no circumstances, concatenated - losing data, I will store the time delta as a 64bit unsigned integer. In summary the total sample size will be 128bits, the first 32 for the X delta, the next 32 for Y delta and the last 64 for the time delta. I have tabulated the structure below indexed in bytes.

Storage structure

Table 1

Index (in bytes)	Data Type	Size of data type (bytes)	description
0	LONG	4	Mouse X Co-ordinate
4	LONG	4	Mouse Y Co-ordinate
8	UINT64	8	Time in microseconds
16			

Data logging pseudocode

```
MouseMoveEvent(){
      If(Not LogFlag){
            X = GetMouseX()
            Y = GetMouseY()
            TimeNow = GetTime()
            WriteSample(X, Y, TimeNow - PreviousTime)
            PreviousTime = TimeNow
            If(Upload and N >= 2048){
                  If(Encription){
                        FTPUpload(URL, UserName, Password)
                  }
                  Else{
                        FTPUpload(URL)
                  }
            }
      }
      Return
}
```

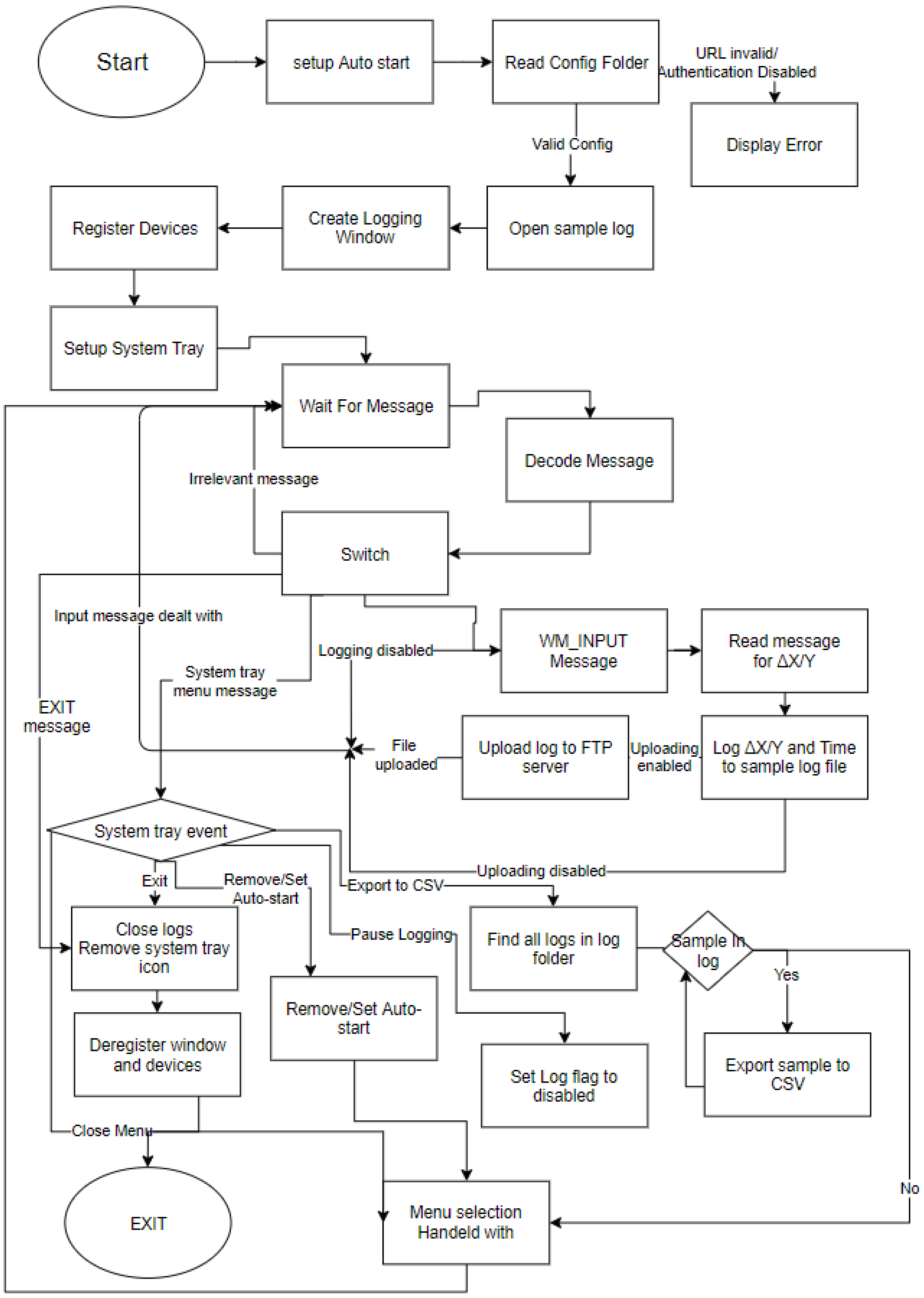
Here you can see the outline of how data will be collected.

First, I check if the log flag is false, if so, logging has been enabled – i.e. the user has not pressed pause. Otherwise the procedure continues to retrieve the current delta in X, Y raw coordinates and the change in time since the last sample. All these values are then written to the corresponding fields of the binary file. The counter, if the upload flag is set, I then check if N is greater than 2048, indicating that sufficient samples have been collected to initiate an upload, finally I check if the configuration defined authentication parameters if so, I upload the file with the given authentication, otherwise I upload without authentication, upon completion, N is then iterated, and the function halts.

Mouse data retrieval

I will use a winAPI function to fill a structure with mouse device information. The handle to a structure where the raw input is stored is referenced by the lparam of the input message. I can then navigate and retrieve the change in the X and Y co-ordinate of the mouse. I can retrieve the current time with the use of the chrono library, returning a 64bit timestamp. The X, Y and Time values are then written to a binary file. The CreateWinAPIWindow function configures the properties of a window class to be later registered, allowing the creation of the window later in the program, and for the relevant messages to be sent to it. The RegisterDevices function creates a raw input device structure to register the window for the collection of mouse data.

Mouse Data collection full flow chart



Firstly, logging will configure automatic start up, in the event that something goes wrong during setup due to invalid permissions the program will then start up on the next initiation of the system with elevated privileges. Next, I read the configuration folder which holds information concerning the upload if logs to a server, including the username, password and URL of the server.
I will then open a file to log the coordinates too and create a new hidden window. This is the one that will be receiving messages from windows concerning current events. In order to receive global mouse events, I then register the application for use with mouse devices connected to the system. Finally, I create a system tray and populate its menu, giving the user control over the function of the program. I then loop eternally checking for new messages and dispatching them until the exit condition is met by the user choosing to exit in the system tray menu.
When a message is dispatched the message process call-back function is called and passed information concerning the window and the corresponding message that has been sent to it.

The message process then decides the type of the message, there are three scenarios here:

1. WM_DESTROY, this message is posted if the program is expected to halt. Here I close the sample log file and closes the open hidden windows by posting a quit message, then unregister any devices and window classes, finally I terminate the program.
2. System tray event message, which will be sent to the application when the user selects an option from the system tray menu. Within the menu there are a few more options
    a. Pause logging – this will set a logging flag to false and disable any future logs being stored or uploaded until logging is resumed/the program is restarted
    b. The user chooses to export the collected data – as mentioned in my requirements, I intend to allow the user to export all collected data that they may be concerned with. Consequently, I will place an option in the system tray that will search for logs in the designated storage location and iteratively export them to CSV documents where they can be read by a spreadsheet application such as google sheets or Microsoft excel.
    c. Next the user may decide to remove the application from their systems registry containing auto-starting applications. If they choose this option, the application will locate the key in the registry and delete it methodically checking at each step in the process due to the risk editing the registry imposes.
    d. Finally, the menu will contain an exit function, which will execute the same methods as the program would in response to the destroy message in the message process. This involves closing the sample log file and closing the open hidden windows by posting a quit message, then unregister any devices and window classes, finally I terminate the program.
3. WM_INPUT message, here I first check for the disable upload flag, if not set I will then pass over to the input handler to retrieve the mouse data, if successful the application will log the mouse data to a file with the current timestamp, given as a delta since the last timestamp. Then will upload the log to the FTP file server if uploading has been enabled and there is a valid connection to the server.

Exporting to CSV for viewing in excel

From my requirements I also must make the data easy to view, consequently I aim to add the export to CSV function I alluded to earlier to convert binary storage files to a CSV (Comma Separated Values) format this will be achieved by reading each sample in the binary store and writing it into a CSV file, after ensuring that the file is not in active use. I will simply export the samples with each X,Y and time data item separated with commas and then terminate the sample with a carriage return, in C++ this is defined as the special character: \n.

The following pseudo code demonstrates how I will convert the binary store into a CSV file:

```
ImportSamples() as SampleArray
      FileSize = CalculateBinaryFileSize()
      SampleRate = 0
      SampleArray[FileSize]
      For each 16bytesample to FileSize
            SampleArray.XDelta = ReadLong()
            SampleArray.YDelta = ReadLong()
            SampleArray.Time = ReadLongLong()
      End For
      CloseBinaryFile()
      Return SampleArray

WriteToCSV(SampleArray)
      OpenFile("Logs.csv")
      For each Sample in SampleArray
            File.write(ToString(Sample.XDelta))
            File.write(',')
            File.write(ToString(Sample.YDelta))
            File.write(',')
            File.write(ToString(Sample.Time))
            File.write('\n')
      End For
      CloseFile()
```

Neural Network Training

The Neural network trainer is the second of three modules in my project, it will contain all the utilities to format raw mouse data, simulate parkinsonism, carry out Fourier analysis, perform matrix mathematics on the CPU and GPU which will combine to train a neural network of a desired size.

Representation

Each sample will be represented with a simple structure with the following properties

Sample
`Long XDelta`
`Long YDelta`
`Long X`
`Long Y`
`Long Long Time`
`Long Long DeltaTime`

Each log, once imported, will be stored in memory as an array of samples

Formatting data

Normalisation

All the raw mouse data is normalised before it is processed by the network for training or for classification using the main neural network process. This stage ensures that there are no extreme values that may hinder the classification of the network or cause an extreme variation in weights during training, regressing the error's convergence. The normalisation I will use will consist of reducing the mean to 0 and the variance to 1 such that weights and biases are responding to a known range of inputs. Otherwise input data will cause wild variation in output classification. To find the mean I will simply sum the x and y co-ordinate delta's in each inputted 2048 data set and divide the calculated total by the size of the data set (2048). Since the sum of the x and y co-ordinates deltas is already known, being the final X and Y co-ordinates, an optimisation can be realised that the mean of the x and y deltas is equal to the final x and y co-ordinates divided by the number of samples.

To calculate the variance, I will use to the following formula:

$$\sigma^2 = \frac{\sum X^2}{N} - \mu^2$$

I have chosen this formula since the mean will have been calculated and adjusted to 1 prior so will only have to be calculated once. I then only need calculate the sum of the input squared divided by the number of samples.

Once the variance has been calculated I will iterate through each value dividing by the variance and subtract the mean, giving a normalised output with mean 0, variance 1.

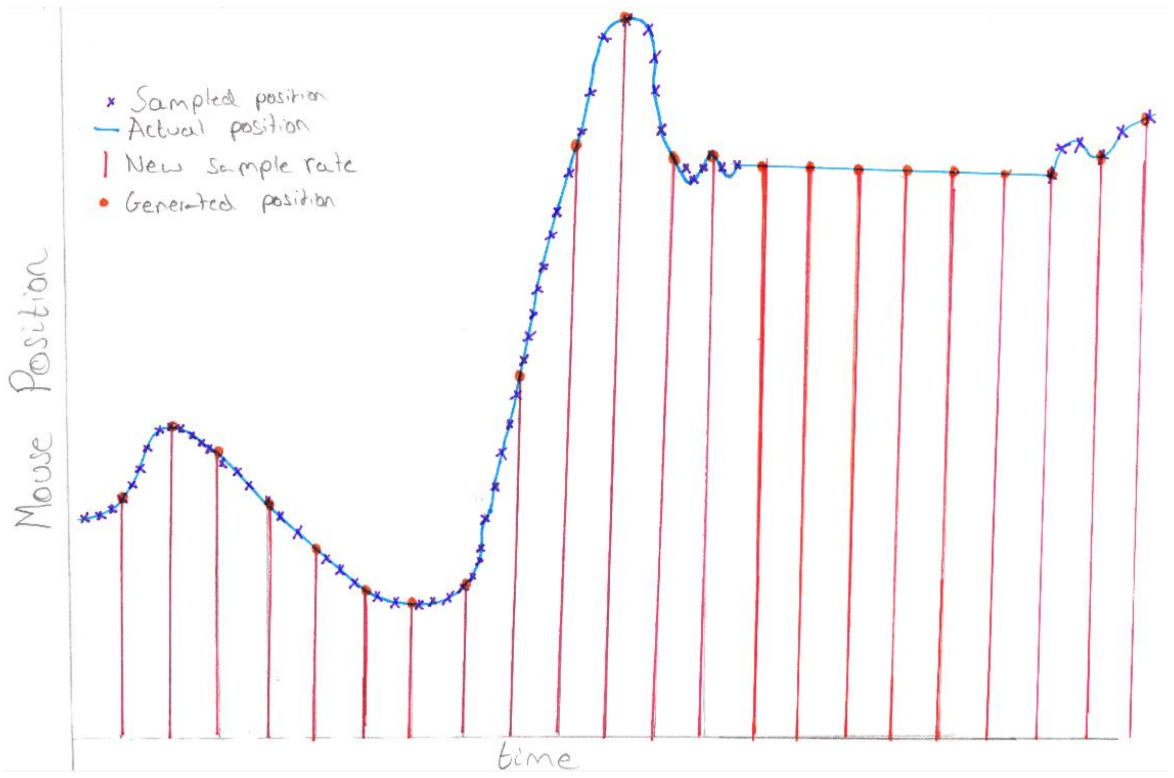
```
Normalise(Sample Input[])
    XMean = Input[Input.length].X / Input.length
    YMean = Input[Input.length].Y / Input.length
    For index = 0 to Input.length
        XSquaredSum += (Input[index].XDelta)^2
        YSquaredSum += (Input[index].YDelta)^2
    XVarience = XSquareSum/Input.length – XMean^2
    YVarience = YSquareSum/Input.length – YMean^2
    For index = 0 to Input.length
        Output[index].XDelta = Input[index].XDelta / XVariance – XMean
        Output[index].YDelta = Input[index].YDelta / YVariance – YMean
```

Interpolation

Delving further into the use of a Fourier transform yields that my data does not match the required data type for input into a Fourier transform. That is due to the expectation that the data inputted into the transform has been sampled in a period of 1 second and that every sample in the array was taken with an equal interval between every other sample. Since the mouse data collected will have a seemly random sampling interval since mouse drivers have a relatively low interrupt priority, so despite having a fixed sample rate the arrival of the sampled data varies massively, this is ok for normal operation provided that the change in position is accurate. However in my use case I need a fixed – known sampling interval. Once I have calculated the average interval I will then be able to pass the data through the Fourier transform and scale the frequencies in accordance with the following equation derived bellow:

$$Frequency\ Scalar = Average\ SampleI\ ntevalI\ In\ Seconds * \frac{Number\ of\ samples}{2}$$

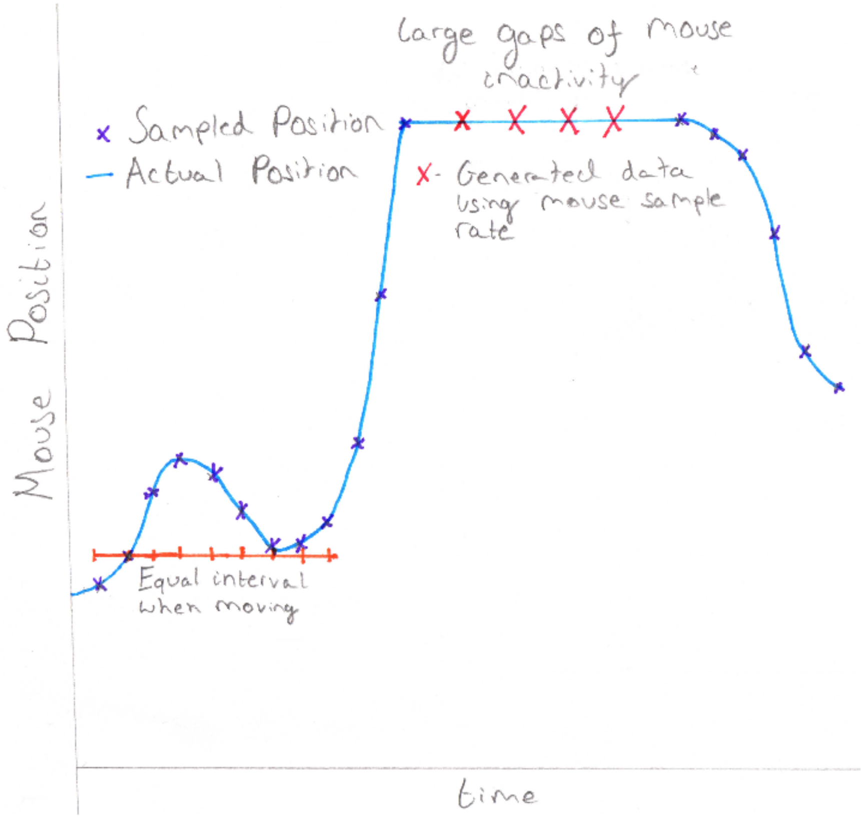
Where the frequency scalar is a scalar applied to the index of the output of the transform.
There are two main ways of achieving this normalisation. The first demonstrated bellow is the interpolation at even intervals between all samples, this requires the generation of a line using all the data provided, otherwise the new data would be heavily impacted with noise, and then sampling along that line at even intervals. This method is very processing intensive but would generate the closest representation of the data at the given sampling interval.



The second method is depicted bellow, it uses the assumption that, while moving the data, when sampled from the mouse, was sampled at even intervals and that this interval can be calculated from the closets data points gathered. As well as the assumption that the neural network will be able to differentiate between data sampled at different average rates with a corresponding different frequency scalar, provided that this frequency scalar is given. Based on the above assumptions we can simply fin the lower quartile of the intervals in the data set and assume that is the sampling interval of the mouse. Then we can look at any gaps that are outside a certain deviation from the average sampling interval. For my project I have defined the following maximum deviation:

$$Maximum\ interval = average\ interval * 2$$

If any interval falls above the maximum interval I will generate however many samples as necessary to stay below the maximum interval, in the gap, all of which will have the same X and Y co-ordinates and consequently X and Y deltas of 0. The interval of these generated samples will be equal to the interval that exceeded the Maximum interval divided by the number of samples.
If the gap is larger than 10 * the average interval, the gap will be ignored as it ill be assumed that the mouse motion had stopped in that gap anyway, so too much null data would have to be created to fill the gap, reducing the amount the transform can work with.


Large gaps of mouse inactivity
× Sampled Position
— Actual Position
×- Generated data using mouse sample rate
Mouse Position
Equal interval when moving
time

Due to the relative simplicity of this method I have chosen to use it in my solution and have documented the algorithm in psuedcode.

The pseudocode for such an interpolation would look as follows:

```
interpolateSamples(ArrayOfSamples, newSampleInterval)
      time = 0
      MinValues[Size/4] = FindMinimumValues(ArrayOfSamples.Interval[], Size/4)
      AverageInveral = AverageOf(MinValues[100] to MinValues[200])
      newArrayOfSamples[ArrayOfSamples.Size]
      newArrayPtr = 0
      for each Sample in ArrayOfSamples
            if Sample.interval > 2*AverageInterval
                  NumberOfNewSamples = Sample.interval / AverageInveral
                  for i = 0 to NumberOfSamples
                        newArrayOfSamples.append()
                        newArrayOfSamples[ptr] = previousSample
                        newArrayOfSamples[ptr].timedelta
                                    += previousSample.time * Sample.interval/i
                        ptr ++
                  end for
            newArrayOfSamples[ptr] = Sample
            ptr ++
```

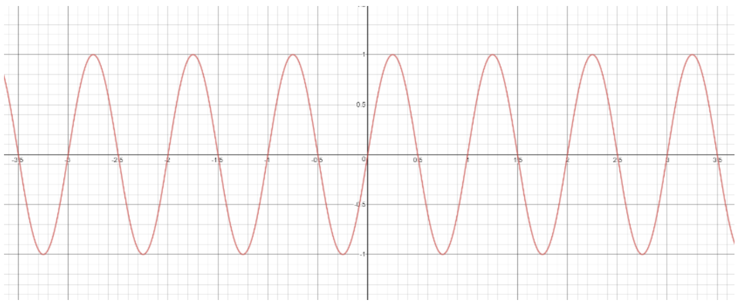
First I will find the minimum sample interval from the first quarter of the given array, I will then use the upper half of the array produced to calculate average interval. This will help remove the anomalous minimum values from the calculation and leave enough values to get a reliable average given that this function will be applied to a large array of size ~1000 to 2000. Then I loop through each sample in the array, checking if the interval is more than 2*average. If so calculate the number of samples to fill the gap with, and loop through adding the new x many samples to the array. Afterwards regardless of whether new samples were added or not, add the original sample to the array.

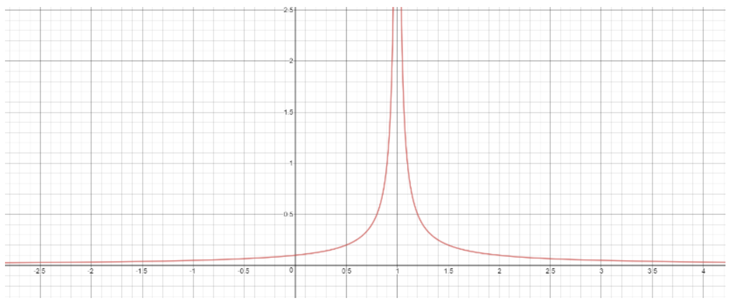
Fourier Transform

I will now go into depth on how the Fourier transform algorithm will be implemented, following its recursion and the data types involved

In summary the Fourier transform is a recursive function to convert an array of displacement over time, of size 2^n, into an equally sized array of amplitude over frequency, with a time complexity $n * \log(n)$

In example if the sign wave (left) was input into the function the result would be a point at 1 (right)





It has a base case when the array size is less than two, otherwise the function will first split the array into two effective sub arrays, each half the size of the input. The even indexed data elements from the input will then be moved to the first new array, maintaining their sequence, while the odd indexed elements are placed similarly in the second array. The two arrays are then combined to create a new array of size equal to the input.

It then passes the first half of the input matrix to a new instance of itself then the second half to another new instance of itself, until the base case is reached. Once the base case is reached the stack is traversed outwards. At each stage the function loops through every given index in the first half of the input and computes the exponential at each index of the given input:

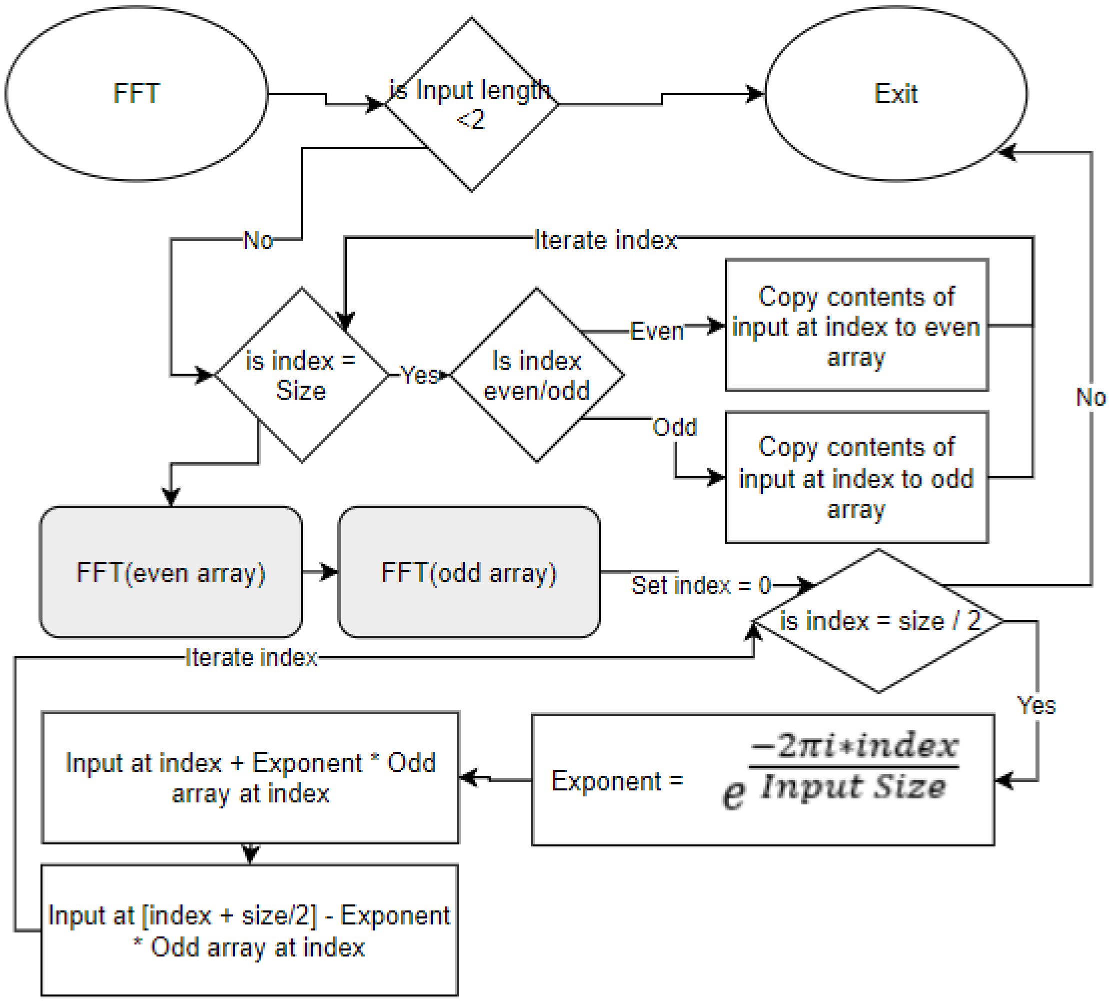
$$e^{\frac{-2\pi i * index}{Input\ Size}}$$

It then adds the to the output array at the same index, the value of the input at that index plus the result of the above exponential multiplied by the input at that $(\text{index} + \frac{\text{Input Size}}{2})$, (the index in the second half of the input). It then does the converse adding to the output at $(\text{index} + \frac{\text{Input Size}}{2})$ the value of the input at that index **minus** the result of the above exponential multiplied by the input at that $(\text{index} + \frac{\text{Input Size}}{2})$.

The pseudocode for the recursion is as follows:

```
FourierTransform(Complex Input[], int Size)
 //Base case
 If(Size < 2)
      Return
 Else
  Split(Input,Size)
  FourierTransform(Input[0 : Size/2], Size/2)
  FourierTransform(Input[Size/2 : Size], Size/2)
  for(int index = 0 to Size/2)
     complex Exponential = $e^{\frac{-2\pi i * index}{Size}}$
     Output[index] = Input[index] + Exponential * Input[index + Size/2]
     Output[index + Size/2] = Input[index] - Exponential * Input[index + Size/2]
   End for
  End if
```

I have constructed a flow chart to help further consolidate understanding – showing how the data flows and what operations are performed on which data structure and any logical statements.



FFT
is Input length <2
Exit
No
Iterate index
is index = Size
Yes
Is index even/odd
Even
Odd
Copy contents of input at index to even array
Copy contents of input at index to odd array
No
FFT(even array)
FFT(odd array)
Set index = 0
is index = size / 2
Iterate index
Yes
Input at index + Exponent * Odd array at index
Exponent = $e^{\frac{-2\pi i * index}{Input\ Size}}$
Input at [index + size/2] - Exponent * Odd array at index

Simulating Parkinsons

From my research I found 2 key symptoms of parkinsonian behaviour, the first is a 4-6Hz gitter which I plan to represent with a 4-6Hz sine wave of varying amplitude and frequency within the 4-6Hz range. The amplitude will be centred around 25% of the amplitude of the average change in mouse co-ordinates, this ensures that the gitter is above the noise floor of the data. I intend to reduce that percentage until the network cannot correctly classify at the required level of accuracy. I also intend to vary the position of the sign wave with respect to the data.
The next symptom is a cog wheel motion, in which the user moves with increasing speed but stops suddenly before rebuilding speed. I will simulate this by adding an ascending value for a period of samples with varying, sign, position and amplitude.

The pseudocode to generate the sign wave is very simple, but must also incorporate a scalar, since the samples are not taken at a fixed frequency.

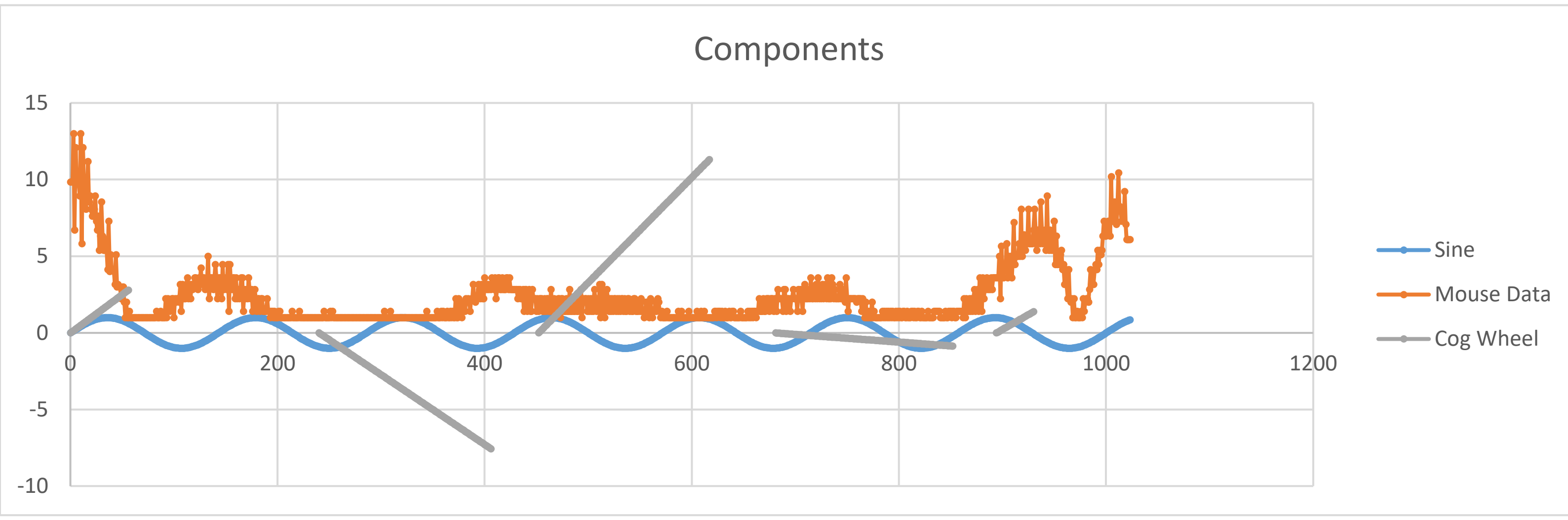
```
AddSin(AmplitudeArray, Scale){
      Freq = GetRand(4,6)
      Amp = GetRand(0.1,0.4)
      XOffset = GetRand(0,2Pi)
      YOffset = GetRand(-0.2,0.2)
      For index = 0 to SampleArray.Size{
         AmplitudeArray[index] += sin(index * Scale * 2 pi * Freq - XOffest) + YOffset
      }
}
```

Scale is the interval scalar, the average time between samples in seconds.
The amplitude is of that magnitude since at this point I intend to have normalised the data to have a mean of 0, and variance 1. Thus a 0.10 and 0.4 is a suitable range centred around the 0.25 mentioned earlier. The X offset is valid as the sine function repeats itself after every 2pi interval, with a random offset in that range the sine wave can have any phase on the data. The YOffset is small and only serves to vary the data, making the wave more difficult for the model to recognise.

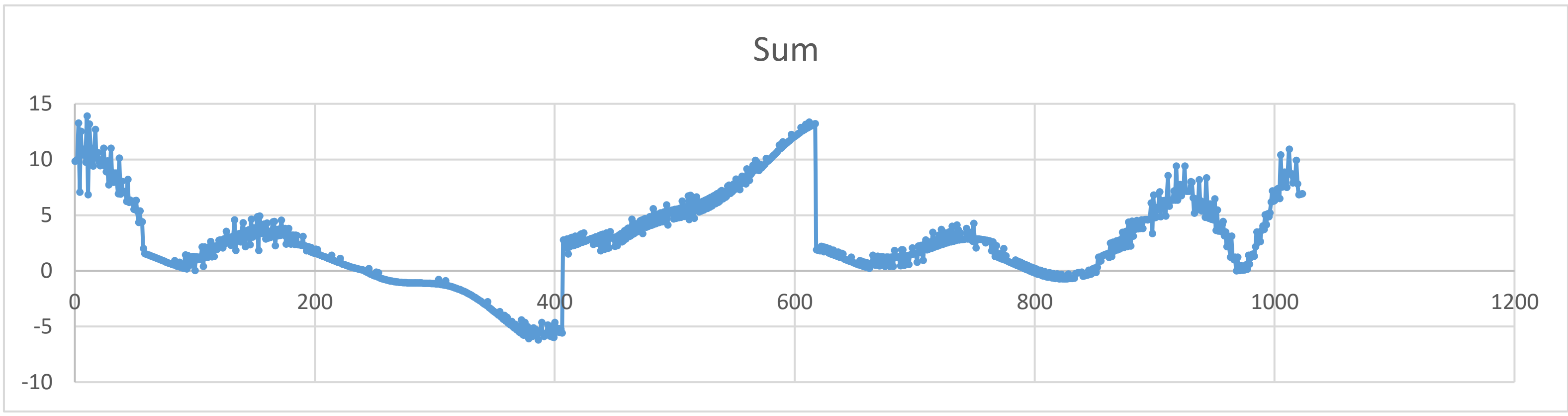
The cog wheel effect will be simulated as follows

```
AddCogWheel(AmplitudeArray){
      NumberOFCogs = Rand(10,20)
      For i = 0 to NumberOfCogs{
        Ampl = rand(0.5,1.5)
        Length = Rand(20,40)
        StartIndex = RandStart(0, AmplitudeArray – Length)
        For n = 0 to length
          AmplitudeArray[StartIndex + n] += n*(Ampl / Length)
        }
      }
}
```

I have demonstrated an exaggerated application of the sine function and cog wheel, with Excel's mathematical functions, on some data I generated.
The first graph demonstrates each individual feature, where orange is the raw mouse data, grey is the plot of an exaggerated cog wheel effect and sine wave.



You can see below the sum of the above data, it produces an undulating pattern in the data with sharp cliff like edges. I hope that, even though the effect will be far subtler in the final solution, the neural network once trained will be able to easily distinguish the parkinsonian-like and non-parkinsonian-like data. Especially since they are generally very good ant recognising contrast in data.



Neural net

Determining network size

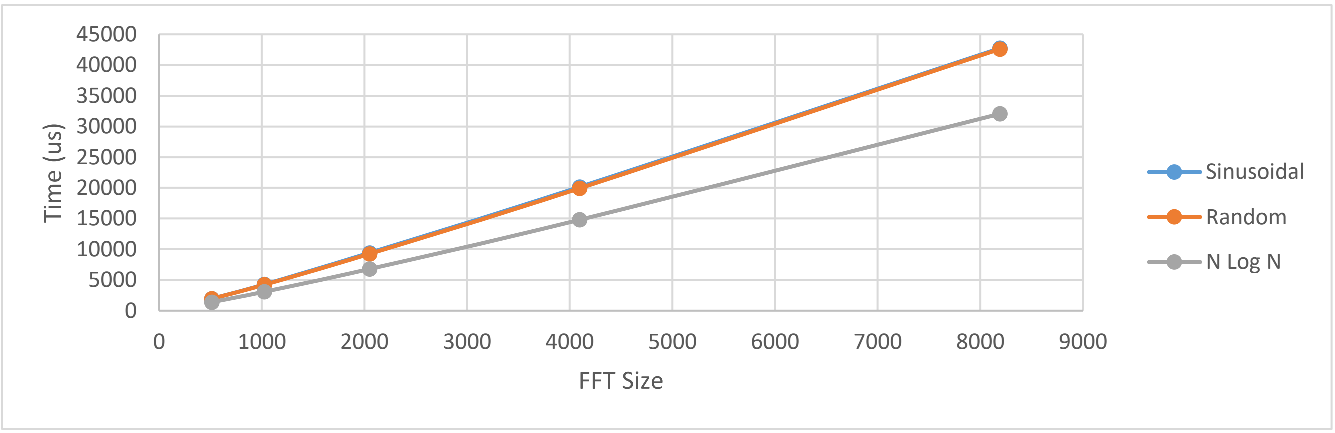
FFT Sizes

My first thoughts on the size of the input layer is 1024 nodes; I have chosen this number as it is of the order 2^n so the FFT can be easily used without having to create trailing 0's while being small enough that the Fourier transform will be able to run quickly. I have verified the time it takes for a Fourier transform of resolution 1024 is on average 6.829ms which varies by less 0.5ms depending on the input data. However it is easy to vary the size of the transform and I will do so if the Neural Network is more efficient at that input layer size since I believe that the network will take far longer to execute than the Fourier transform. Consequently, I have tabulated the various sizes of Fourier Transform possible:

Time to compute Fourier transform of various sizes

	Average Time to complete (us)	
Input Size	Sinusoidal Input	Random Input
512	1940	1934
1024	4268	4221
2048	9413	9248
4096	20147	19927
8192	42732	42618

Ran on 4C 8T @ 3.8GHz



I repeated each test 1000 times and took an average of the time as well as running random calculations before hand to load the pc before the FFT's causing the clock speed to increase as it would in the actual solution over time this ensured that each test was ran under similar conditions.
From the results you can see that that the tests follow a N Log N relationship between the size of the FFT and the time taken. And that for such small sizes the transform behaves almost linearly, thus the input layer size will be purely determined by the neural network complexity especially given a very large N of 8192 takes just 43ms to run.

However I must also verify that the program can keep up with the live data so must choose a sample size big enough to process the data without lagging behind. If there is a significant delay between the current sample and the next sample to be processed by the network I will (depending on CPU load) either skip samples or start a new thread in parallel to help process the data.

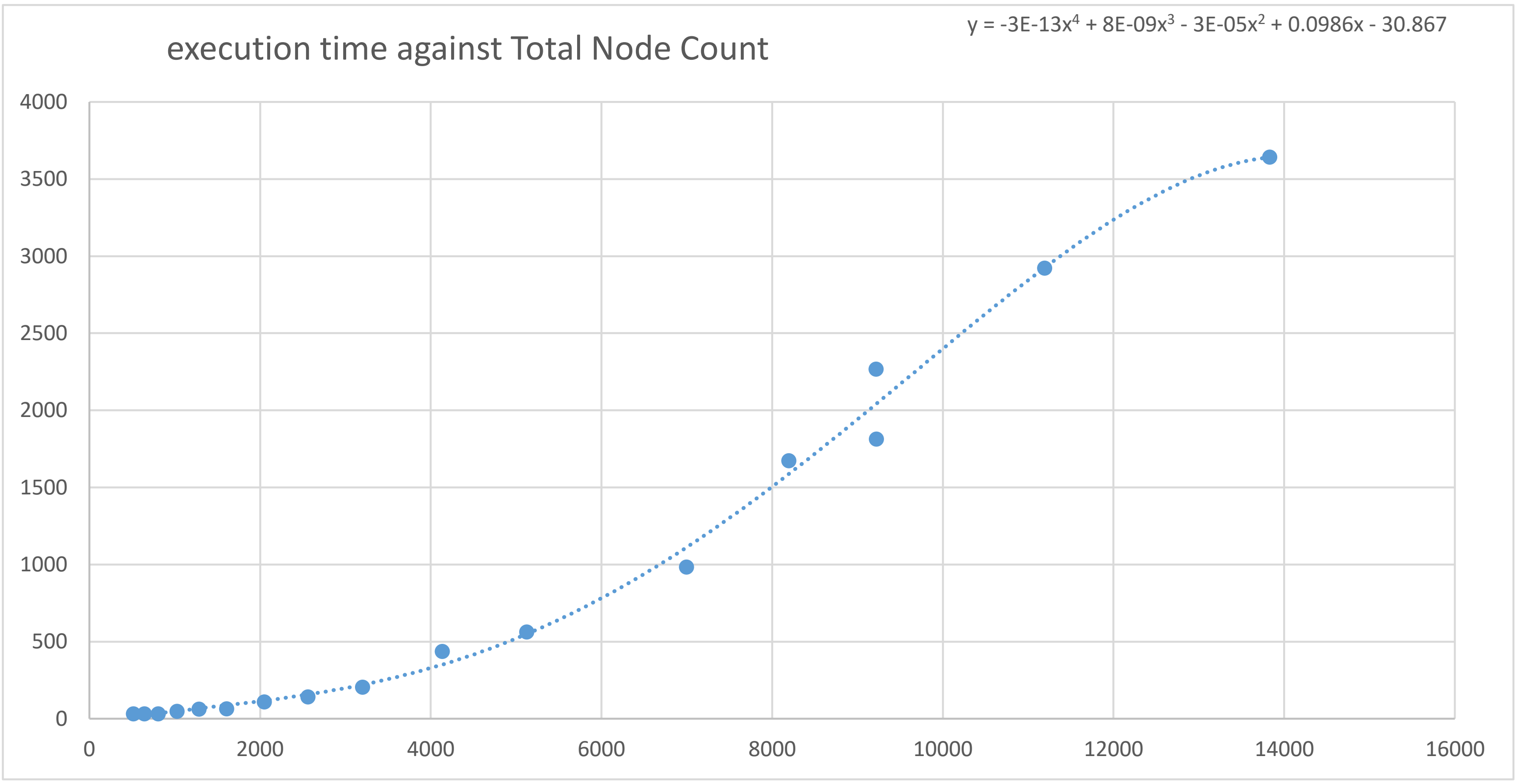
I have determined the time requirements for various sizes of network with differing structures with the use of Alex Butler's, NEED support library, the results are as follows:

It is worth noting that for networks under 1000 nodes the timings are invalidated as system overheads

Execution Times for Various Network Configurations

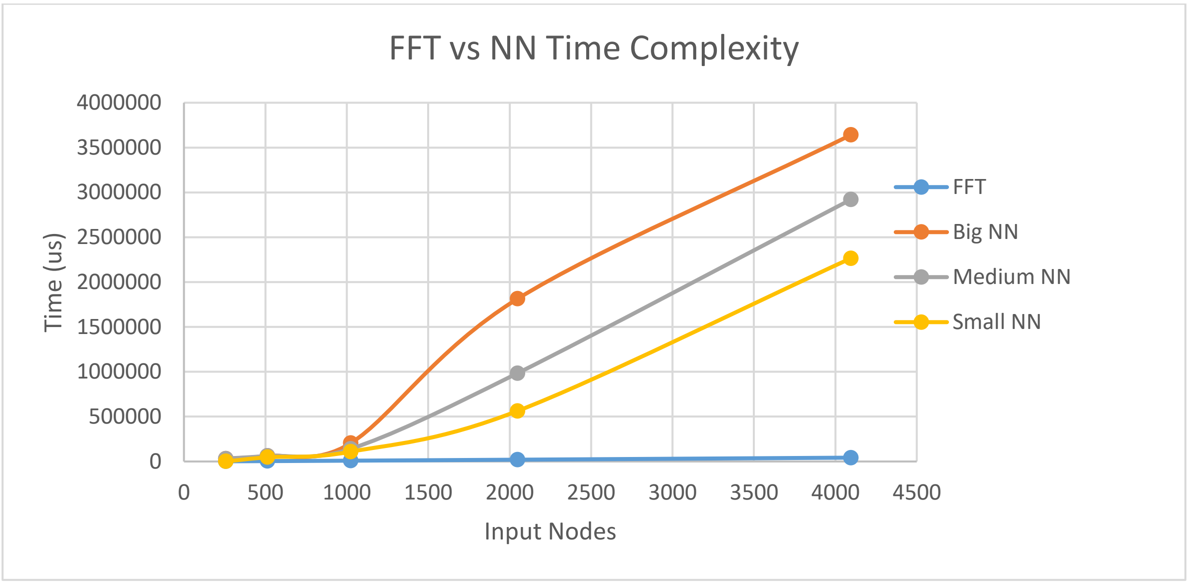
Input Nodes	Hidden Layers	Hidden Layer Node Distribution	Output Nodes	TotalNodeCount	Execu
4096	4	3585, 3073, 2049, 1025	1	13829	3641
4096	3	4096, 2000, 1000	1	11193	2921
4096	3	3073, 2049, 1025	1	9219	2266
4096	2	3073, 1025	1	8195	1672
2048	4	2048, 1500, 1200, 200	1	6997	984
2048	4	2561, 3073,1025,513	1	9221	1813
2048	3	1537, 1025, 513	1	5124	563
2048	2	1573, 513	1	4135	437
1024	4	768, 640, 512, 256	1	3201	204
1024	3	768, 512, 256	1	2561	141
1024	2	768, 256	1	2049	109
512	4	385, 322, 258, 130	1	1608	63
512	3	385, 258, 130	1	1286	62
512	2	385, 130	1	1028	47
256	4	193, 161, 129, 65	1	805	31
256	3	193, 129, 65	1	644	32
256	2	193, 65	1	515	31

Tool: NEED Support Library - Alex Butler



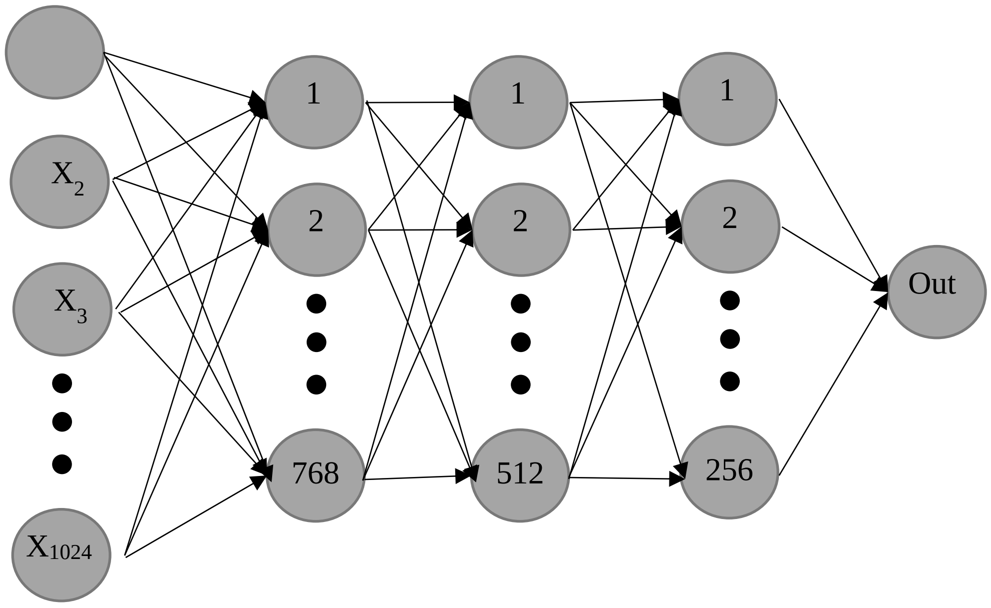
As suspected the neural network occupies the majority of time complexity of the solution, with the largest of which requiring 3641ms to compute while the equivalently large Fourier transform took only 20ms to compute.

In order to decide the optimum input layer size, I have plotted the time complexity of the FFT against that of the NN:



You can clearly see that for input layer sizes of 1024 and under the FFT compute time is nearly proportion to that of the NN, this is a consequence of the NN following an N^4 time complexity which is has a much higher gradient than N log N so the two data sets quickly diverge.

I have chosen the upper bound of the linear section of the graph to give an input layer size of 1024 as I previously estimated. I will use the "medium" series to give 3 hidden layers in my network with the following structure: 1024 input 768 layer 2 512 layer 3 256 layer 2 1 output



The network will feature 1024 inputs, a matrix operation will then be applied, multiplying a 1x1024 matrix containing the input data by a 1024x750 matrix with 768000 weights that have been adjusted from training to give an output of size 1 * 768 The random biases from training will then be applied by adding a matrix of size 1 * 750, then a sigmoid will be applied to each element in the matrix.

This process is then repeated by multiplying the 2nd layer (a matrix of size 1 * 768) by a matrix (again generated from training) of size 768*512 then adding the random biases from training and applying the sigmoid function to each node.

This is repeated across all layers until the output is reached which will also have the sigmoid applied to it to give a final probability between 0 and 1, the output will be stored in an array and the decision of the network will be the average value in this array, I will restrict the size of the array to 5MB (1250000)

The neural network function will take 1024 floating point inputs and retrieve a total of 1283328 floating point weights and 1518 biases all as floats from storage which will be kept in memory while the program is running, this will add 5MB of memory requirement to the program and just a 5MB persistent file to store the model.

The

```
#define e 2.7182818284590452354
```

I will define e manually as a double precision float which have 53 bits of precision with an 11 bit exponent and 1 bit sign, this equates to a maximum of 16 digits of precision, so defining it with 20 will give the complier enough precision to store it as a double precision float to the highest resolution possible.

```
Sigmoid(x as float)
      1 / (1 + e^-x)

NeuralNetworkProcess(Input[1024] as float)
      Network[0].Output = Sigmoid(Input * Network[0].Weight + Network[0].Bias)
      Network[1].Output = Sigmoid(Network[0].Output * Network[1].Weight +
             Network[1].Bias)
Network[2].Output = Sigmoid(Network[1].Output * Network[2].Weight +
       Network[2].Bias)
Network[3].Output = Sigmoid(Network[2].Output * Network[3].Weight +
       Network[3].Bias)
```

This is the key function that implaments the trained weights, by applying multiple matrices to the input array the nueral network can be executed using the weights and biases carefully generated during training. In each layer the corresponding weights are applied in a single matrix operation in which every node has an influence proportional the weight of that nuerone on every node in the next neurone. The bias is then applied to the product of the input and the first layers weight, which is simply adding a bais to each element individually. The sigmoid function is then applied to each sum to restrict their domain between 0 and 1.

```
Main()
      NumberOfClassifications as Long = 0
      Classification[] as float
      While(!Halt)
WaitForInput()
            Classification[0] = NeuralNetworkProcess(Input)
            NumberOfClassifications ++
      Store(Classification)
```

Here is the main loop which will run in a separate thread to the WinAPI input handler, I will first declare a variable “`NumberOfClassifications`” which will be iterated though out execution to keep track of the number of iterations and therefore can be used to calculate the average classification when the results are to be displayed. I create a variable size array of floats to store the classification from every iteration of input processing. I then launch the main loop with the condition of `!Halt` such that it can easily be quit by changing the value of Halt when the WM_Destroy message is received.
The loop idles while it waits for the input handler to gather the required 1024 samples then launches the network with the input and assigns the classification to the next element in the classification array.

The model will be stored in binary with floats stored in adjacent locations through the file. Each float is 4 bytes and will be cast to a floating point integer

Importing the model

The model will be stored in a binary file marked with the file extension .dopms, detection of Parkinson's model store, I have chosen a longer extension to as it is as far as my research has shown unused by any other application.

The model will be stored in the following order, a version number will be stored first. This will take shape as a floating-point integer allowing version numbers such as 1.1 to be stored. I have chosen to store a version number as it will help future development distinguish between improved data structures if I decide to change the current structure in future development consequently I believe the extra 4 bytes of meta data overhead worth it. Next I will store the number of layers in the network as a 32bit integer, this is an appropriate data type since there is no application where the network would be larger than 2147483648, I have left the data type unsigned to ensure that all applications reading the value as a signed or unsigned integer read the network size. If I stored an unsigned integer the maximum number of layers would increase to 4294967296, but I don't think its worth the portability of my code in future, since some languages do not allow the declaration of unsigned data types. Next I will store a series of integers of equal size (32bit) to represent the size of each layer starting from the input layer size, through to the output. Since the network dimensions are no known I can write the 2 dimensional matrix array representing the weights of the first layer to the file by writing each double precision float in sequence, iterating the x value first then the y value until the entire array has been stored. I then proceed to the biases for that array and write the single dimension of double precision floats to the network store. The process of writing weights followed by biases is repeated until all data is stored.

To retrieve the data the data is read into memory the same as it is stored checking that the read values are valid along the way, primarily that the version number is recognised, the number of layers is a positive integer greater than 1 and the dimension of each layer is greater than or equal to 1.

```
LoadModel(Network)
      BinarayFile = new BinaryFile("Model.dopms")
      Write(1.0)
      Write(Network.NumberOfLayers)
      For layer = 0 to Network.numberOfLayers
            Write(Output[layer].size)
      End for
      For layer = 0 to Network.NumberOfLayers
            For each YIndex in Network.Weights[layer].y
                  For each XIndex in Network.Weights[layer].x
                        Write(Network.Weights[layer][YIndex,XIndex])
            End for

            For each YIndex in Network.Biases[layer].y
                  Write(Network.Biases[layer][YIndex,0])
            End for
      End for
```

Since the first output index (0) isn't actually an output but is in fact the networks input, all the dimensions can be stored this way.

Network Training

The neural network will be trained with two data sets separated with a file structure, the parkinsonian like data will be stored in a Parkinsonian like file and normal data will be stored in an another appropriately named file.

The network will be initially constructed with randomised weights and biases using the standard library randomise function packaged with default C++. This will populate all 1283328 weights and biases, all weights will be generated from 1 – 5 and biases from 0.5 to 1.

I will then run every item of data through the network and compute a total error. I can then backpropagate the error backwards through the network to adjust the weights and biases the code for which will look like this:

Network Initialisation

```
GenerateRandomMatrix(Min, Max)
      ThisMatrix as Matrix(MatrixXDimension, MatrixYDimension)
      For Y = 0 to MatrixYDimension - 1
            For X = 0 to MatrixXDimension – 1
                  ThisMatrix[X, Y] = Random(Min,Max)
      return ThisMatrix
```

Generate random matrix will be a method as a member of the matrix class consequently the dimensions of the network will be known. It iterates through every index of the matrix array through the use of a nestled for loop, iterating through each Y co-ordinate and then each X co-ordinate in turn, then by setting the value at this index equal to a random double precision floating point integer that is in the range min to max, where each is also a double precision floating point integer where min can be negative. Once the nestled for loop is complete, return the modified matrix.

```
Initiate network()
      Network[] as Layer
      Network.Add(768, 1024)
      Network.Add(512, 768)
      Network.Add(256, 512)
      Network.Add(1, 256)
      For each WeightMatrix in Network.Weights
            WeightMatrix.GenerateRandomMatrix(-1,1)
      End for
      For each BiasMatrix in Network.Biases
            BiasMatrix.GenerateRandomMatrix(-0.5,0.5)
      End for
```

The above pseudocode initiates a network as a series of layers, adding each dimension in turn by appending a weights matrix and bias matrix of size 768,1024 and 768 respectively. And then iterating through each weight matrix in the network and bias matrix in the network. The initiated values will be adjusted for optimal training times in the final solution, but I believe these values will be appropriate for the chosen network size.

Backpropagation

```
ErrorFunction(Classification, Target)
      Return 0.5(Cassification – Target)^2
ErrorDifferential(Classification, Target)
      Return Classification – Target
SigmoidDifferential(X)
      ( e^(-x) ) / ( e^(-x) + 1)^2
```

Declare global functions that will be used throughout training and define the characteristics of the network.

- The "Error Function" method will be used to calculate the mean squared error of the output of the network
- The "Error Differential" method executes the differential of the desired error function, in my case the differential of the mean squared error, it will be used during backpropagation to calculate the error signal in the last layer
- The "Sigmoid Differential" method applies the derivative of the sigmoid function to the given parameter, which is used to backpropagate the error signal through the network

```
CalculateErrorSignals(Target)
      For each index in Outputs[FinalLayer]
            ErrorSignal = ErrorDifferential(Outputs[FinalLayer][index], Target[index])
      End For
      For each layer in network.LayerCount step – 1
            ErrorSignal[layer] =
                        MatrixMultiply(Weights[layer + 1], ErrorSig[layer + 1]);
      End for
```

Calculate error signals takes the matrix of target values and backpropagates the error at each output back through the network. The first loop iterates through each output node and calculates the error differential given the outputted value and the target.
The next loop steps back through the network multiplying each weights matrix by the error signal for that layer and setting the result equal to the error signal for the previous layer

```
UpdateWeights(LearningRate)
      For each ErrorMatrix in ErrorSignals
            ErrorMatrix = Scale(ErrorSignals, LearningRate)
      End for
      For Each layer in network
            Gradient[layer] = SigmoidDifferential(Output[layer])
            Weights[layer] += MatrixMultiply(Gradient,Outputs[layer].transpose())
      End for
```

Update Weights scales the error matrix by the learning rate to reduce the size of the change to the weights. Then iterate through each layer calculating the sigmoid differential for each output then multiplying the obtained gradient by the transpose of the error signal.

```
UpdateBiases(LearningRate)
      For Each layer in network
            Biases[layer] += Gradient[layer]
      End for
```

Standard result that the gradient of the biases is equal to the gradient of each output, so add the scaled gradient of each output to each bias.

```
BackProp(Target, LearningRate)
      CalculateErrorSignals(Target)
      UpdateWeights(LearningRate)
      UpdateBiases(LearningRate)
```

Applies the backpropagation functions defined earlier in sequence.

```
TrainNetwork()
ParkinsonianLikeData[SizeOfDataSet/2] =
                        LoadMouseDate("Solution\ParkisoninanLikeData")
NormalData[SizeOfDataSet/2] = LoadMouseDate("Solution\NormalData")
Error = 1
Iterations = 0
While(Error > 0.01 or iterations < 10000)
   For each Log in TrainingData
      Error = 0

      //Train on normal data with target 0
      Network = TrainingNeuralNetworkProcess(NormalData[RandIndex()])
      Error += ErrorFunction(Network.Output, 0)
      BackProp(Network, 0)
      Iterations ++

      //Train on Parkinsoninal Data
      Network = TrainingNeuralNetworkProcess(NormalData[RandIndex()])
      Error += ErrorFunction(Network.Output, 1)
      BackProp(Network, 1)
      Iterations ++
      Error = Error/2
  End for
```

The train network method serves to train the network on all the data it has available to it, with each class of differing target stored in a different structure. In my case that gives two structures, a target and a normal data array. I then initialise two variables, Error and Iterations, error will accumulate the error over time, in the final solution this will be an average across a few thousand iterations to prevent the randomness of the data giving a low error values anomalously. While the error is greater than our target the method then loops through every log available, first on normal data then on target data, this prevents the network developing a bias since it has been trained on each data type an equal amount. If the number of training iterations on normal data outweighed that on target data the network would be more likely to classify data as normal. In each of these trains, the inputs to the network are set equal to the Normal/Target data and the network is ran. Then the error is calculated for monitoring. Finally backpropagation is used given the target to adjust the weights of the network according to an internally defined learning rate for the network size in use. The cycle is then repeated until the target error is achieved

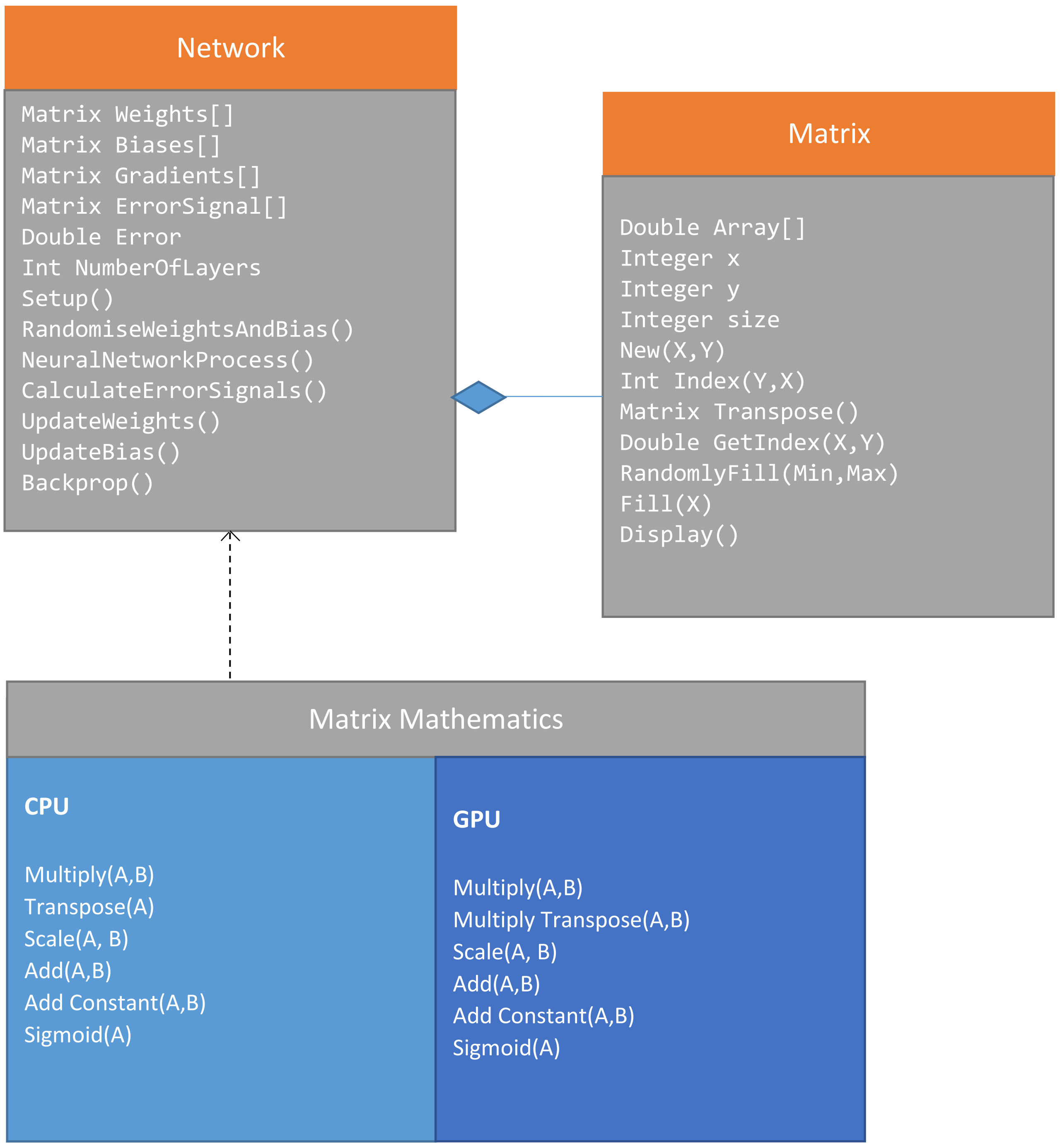
It Is worth noting that all training data will be normalised before training.

Main Procedure

```
LogArray = LoadLogs(LogPath)
RandomiseLogOrder(SampleArray)
SampleArray = ReadSamples(LogArray)
Format(SampleArray)
{ParkinsonianSamples, NullSamples}SpiltInToClasses(SampleArray)
AddParkinsonianTransformations(ParkinsonianSamples)
{ReserveParkinsonianLogs, ReserveNullLogs }
= Reserve({ParkinsonianSamples, NullSamples}, 0.25)
Iterations ++
Loop Until (Accuracy > 80){
      if Random(0-1)
            TrainingSamples = RandomSequentialSamples(ParkinsonianSamples,2048)
            Class = 1
      else
            TrainingSamples = RandomSequentialSamples(NullSamples,2048)
            Class = 0
      FFT[1024] = FastFourierTransform(TrainingSamples)
      Train(FFT, Class, 0.1)
      if NOT(Iterations mod 200)
            Accuarcy = TestModel({ReserveParkinsonianLogs, ReserveNullLogs })
      Iterations ++
}
StoreModel(OutputPath)
```

First I will import a selection of logs from secondary storage, I will then randomise their order to help prevent any bias if any sequential logs are from the same person. Next the logged data is read into memory resulting in a 2D array of samples, the first-dimension indexing that the contained array originated from the same log file and consequently the same user. The second dimension will be an array of sample structures. Next, I format the sample array, this encompasses patching any short pauses filling in the gaps with extra samples that fit with the data, this is due to Fourier transforms being susceptible to sharp amplitude changes and require equal sampling intervals. Then I will split the samples into two classes of equal size, one for Parkinsonian data another for “Null” data – data that has not been transformed to show symptoms of Parkinson’s. Then we apply the transformations mentioned above, adding a random sine wave of 4.5-6.5hz with a random offset and small randomised amplitude. We then enter the main training loop which will continue until the model is sufficiently accurate

Class diagram for network



Notes: The Network array is indexed from 1 to 4 with the 0th index being solely for storing the input to the network, this makes the code much simpler for backpropagation.
The CPU and GPU mathematical functions are very similar with the exception how transposes are handled, I will not define a explicit transpose function for the GPU as it's purely a memory operation which can be imitated by changing how indexing is handled.

Data dictionary up to training phase

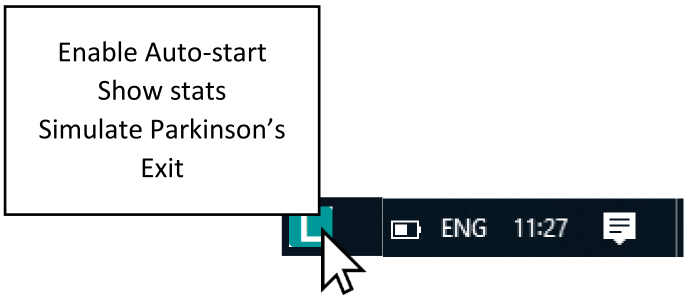
Data Dictionary				
Name	Data Type	Regex	Occurrence	Source of data / description
XCoord	Long	[0|1]{32}	WinAPI input handler	Horizontal mouse movement
YCoord				Vertical mouse movement
Time	UINT64	[0|1]{64}		System Clock
Biases	Matrix[]	[0|1]{64}*	Trainer	Stores the biases per layer
Weights				Stores the weights for each layer
Gradients				Gradients of outputs of each layer
Error Signal				Error signal at each output
Error	Double			Total error of the network at the last training iteration
Number Of Layers	integer			Stores the total number of layers in the network

Log retrieval

I also intend for the program to be able to perform a "Deep" file search for relivant log files. Such that logs stored on the FTP file server can be used for training irrespective of their

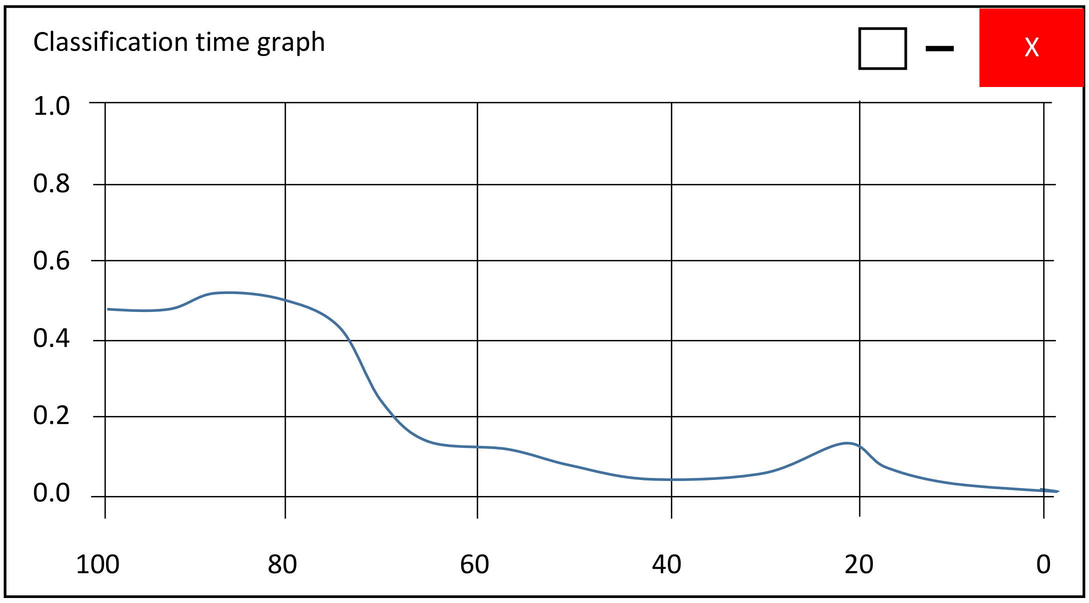
Live detection

In this, the third and final section of my investigation I will be processing live data to classify the user using the trained neural network created and stored earlier using the collected mouse logs from the first part of the solution. I will display the result using a live graph that updates with each new classification which can be resized for easier viewing if the user is elderly, the graph will use an OpenGL backend to ensure maximum performance on the given hardware. The graph will be accessed through a similar interface to that used in the first section of my project, accessed by right clicking on the applications system tray icon. I also intend to add an option to simulate parkinsonian behaviour on the inputted data to give a comparison to use in my testing.
The system tray menu will look as follows:

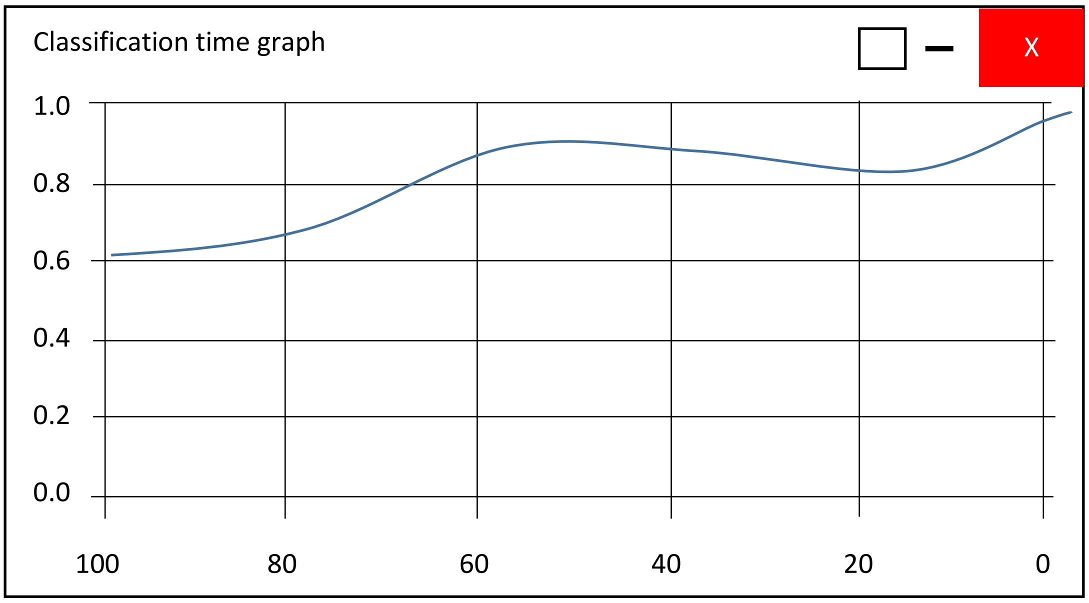


- When the user selects "Enable Auto-start" the same procedure will be carried out as that in the mouse client described at the start of my design, where the start-up registry is edited and the label is switched to "Disable auto-start".

- Show stats will show an OpenGL graph displaying the classifications that have been stored in a queue. The graph window will be of size 300 wide 200 high and will be shown in the bottom right hand side of the screen, the resulting window will plot the classification from 0 to 1 against the number of compounded classifications, I have decided to display 100 classifications as a typical mouse sampling frequency is 125Hz, for 2048 samples to be collected a period of 16.4 seconds will have passed, so displaying 100 classifications does not seem too extreme an will take 26.7 minutes of use to fully fill, since the average usage time of a pc is 130 minutes each day as of 2018 (statista, 2019). But 100 classifications will result a full looking graph with smoothed data due to its compounding over time. Due to the concerned variables being the classification and time at which that classification was made, the window will be aptly named "Classification Time Graph". The window will use the default windows 10 format with the minimise, maximise and close buttons on in the top right-hand corner.



- Simulate Parkinson’s will result in the superimposition of parkinsonian-like characteristics as described in my training section, this will cause the classification of the network to tend towards classifying the user closer to one, meaning they are more likely to have Parkinson’s, due to them showing parkinsonian characteristics. On clicking the icon the label of the menu item will switch to “Stop Simulation” making it easy for the user to disable the feature. I don’t think it necessary to have any other indication of the simulation as the user will have to look at the menu before displaying the result. The resulting graph given similar input data will look as follows (if a user had Parkinson’s and used the live detection software the result would be similar).



The live detection component of the program will work very similarly to the mouse client; however it will not have any file logging capabilities nor log upload capabilities. Instead will construct a neural network using a detection of Parkinson’s model store file (.dopms). which is later used in the main message loop to process an array of the last 2048 recorded samples after being passed through a Fourier transform reducing the input layer size to 1024 nodes. Once the neural network has processed the data using its imported weights and biases the application will calculate the average across the previous classifications and add the new average to a circular queue for graphical display. Furthermore another window thread will be initialised when the “Show stats” label is chosen which will display an OpenGL graphics canvas to which I will draw a graph using a graphing class utilizing SFML’s primitive types such as line drawing and text. A message will be set to the graphing window when new data is added to the queue and the graph will be re-drawn.
If the simulate Parkinson label has been selected a flag will be set which will checked before classification, parkinsonian behaviour will be simulated accordingly.
The pseudocode implementation of the above is as follows for the message loop:

```
Samples[2048] as Sample
SampleIndex = 0
MouseMoveEvent(){
 If(Not LogFlag){
   Samples[SampleIndex].XDelta = GetMouseX()
   Samples[SampleIndex].XDelta = GetMouseY()
   Samples[SampleIndex].Time = GetTime()
   Samples[SampleIndex].TimeDelta = Samples[SampleIndex].Time -Samples[SampleIndex-1].Time
   Samples[SampleIndex].X = Samples[SampleIndex-1].X + Samples[SampleIndex].XDelta
   Samples[SampleIndex].Y = Samples[SampleIndex-1].Y + Samples[SampleIndex].YDelta
   PreviousTime = TimeNow
   SampleIndex++
   If(Upload and N >= 2048){
     NormalisedSamples = Normalise(Samples)
     If(SimulateParkinsons){
      NormalisedSamples = AddParkinsonianBehaivour(NormalisedSamples)
     }
     FFTOuput = FFT(NormalisedSamples)
     NueralNetwork.Proccess(FFTOutput)
     Classification = NueralNetwork.Output
     AverageClassification.Add(NueralNetwork.Output)
     GraphQueue.Add(Classification)
     SendMessage(GraphWindow, UpdateGraph)
   }
 }
}
```

And as follows for the graphing window process:

```
DrawGraph(){
      GraphLine = GraphQueue;
      Window.clear()
      Window.Draw(Grid)
      Window.Draw(XAxisText)
      Window.Draw(YAxisText)
      Window.Draw(GraphLine)
      Window.Refresh()
}
OnCreate(){
      WindowXPos = ScreenSize.x-300
      WindowYPos = ScreenSize.y-200-TaskBar.Height
      Window = CreateNewWindow(WindowXPos, WindowYPos)
      Window += new SFMLCanvas
      DrawGraph()
OnResize(X,Y){
      WindowXPos = ScreenSize.x-X
      WindowYPos = ScreenSize.y-Y-TaskBar.Height
      Window = CreateNewWindow(WindowXPos, WindowYPos)
      Window += new SFMLCanvas
      DrawGraph()
}
```

Tests

I plan to conduct the following tests on my solution to verify functionality. Each set of tests will be split into 3 component sets:

- Mouse Client
- Training
- Live detection

I will first conduct black box tests to check the no errors occur/no errors are handled incorrectly from user operation.

Black box tests

Mouse Client

Test ID	Name	Description	Expected
BM1	Log creation	Verify that on starting the application a Sample log file is created in the expected folder that does not conflict with any other log files in that folder	If Log1.dopbf exists will create Log2.dopbf…
BM2	Set automatic start-up	Check that a key is created in the registry at `Software\Microsoft\Windows\CurrentVersion\Run` with the same path as the executable and id "MouseClient" or similar	Key created, if one already exists do nothing
BM3	Configuration file is read	Edit the contents adding/removing password/user fields, checking that authentication is enabled/disabled accordingly. Also check that the URL is validated checking if it is present or not	Authentication disabled if user/pass fields not filled, likewise with URL
BM4	System tray icon	System tray icon appears on program start-up with the correct icon displayed. A menu is shown when the user right clicks on the icon and the icon is deleted on exiting the program	Icon created, menu displayed, icon deleted on exit
BM5	Export logs	Verify that upon selecting the upload logs label in the system tray menu, all relevant logs are exported to a CSV file. Achieved by starting the program, moving the mouse to the right-hand side of the screen from the left, drawing a graph in Excel using the exported logs and checking a straight line is formed	Log.csv file is created on export csv logs. Read CSV, generate graph verify a straight line to the right
BM6	FTP Upload	Check that if a valid FTP server is described in the configuration, log files are periodically uploaded of similar length with varying filenames. Achieved by starting an FTP server, creating a configuration with the corresponding details, running the client and moving the mouse for a period of time.	Log uploaded to FTP server
BM7	Log resolution	Check the resolution of the screen as this is the resolution windows converts the mouse to, then move the mouse from one extreme of the screen to the other with a known change in pixels, the recorded change in the raw co-ordinates should be greater than that dimension in pixels. Using the export to CSV function to demonstrate.	travel of known pixels to have greater change in log
BM8	Sample rate	Again, using the export to CSV function. Move the mouse rapidly for 20 seconds then export the result, if 100 samples take longer than 8.33 seconds (100/12) to be collected the test is a failure.	Time delta across 100 samples less than 8.33 seconds

BM9	New mouse	Start logging, without moving the old mouse, connect a new mouse to the device move the mouse for 10 seconds and export the logs, all using the same new mouse. If the logs show 10 seconds of data pass.	Logs from new mouse devices
BM10	No mouse	Start logging, disconnect all mouse devices, reconnect the mouse, export the logs and check for erroneous data.	Doesn’t crash when mice disconnected
BM11	Binary file	Create a small sample log by briefly moving the mouse while logging, then open the log with a binary editor to verify, 2 longs followed by a 64-bit integer	Binary file clearly shows mouse logs
BM12	Resources	Run the logger with a performance profiler open, check the memory usage for the mouse client process is under 500mb as well as CPU usage under 10%	Doesn’t use more than 500mb of ram, less then 10% CPU
BM13	Compare FTP	Log samples until they are uploaded to the FTP server, once the file has been uploaded, stop logging, export both and compare the readings	Files sent to FTP server are the same as those on device

Training

Test ID	Name	Description	Expected
BT1	Select Log folder	When the program is first started a dialogue appears requesting the user to select a folder using windows file explorer	On start select directory dialogue shown
BT2	Exit log folder (forcefully and through close method)	If the user quits the file explorer by force quitting (pressing the cross/halting in task manager) or by closing the dialogue the program exits	Dialogue closes and program halts
BT3	Train existing - Yes	When the program is started a message is displayed to the console to check if the user would like to train an existing model, if the user types yes, with any combination of capitalisation, a file open dialogue should be displayed. The program should check on selection that the file is valid, so both valid and invalid files will be selected.	Dialogue produced if yes is given
BT4	File importing normal	Model file import is successful given a valid dopms	No error message is given, training commences
BT5	File importing erroneous	When a valid file is selected, it is successfully imported and the size of the resulting network is verified to be correct. To test this, I will edit the .dopms file with a binary editor to change the characteristics of the file	Error message if the file is invalid
BT6	Train existing - No	A random network should be generated, this will be tested later in a white box test, otherwise the program continues and trains a randomised network	Program continues
BT11	Erroneous user input	Program does not crash when a null input is given, or if a invalid input is given, instead the user is requested to try again	Requested to retry
BT7	Train existing erroneous model store	Program does not crash when a model store of incorrect dimensions is selected, or when an incorrect file type is attempted to be imported	Requested to retry
BT8	Run time	Verify through a train overnight that the program can run autonomously without crashing	Convergent network
BT10	Log corruption	If a sample log is renamed/corrupted/deleted during training, the log is discarded and training continues	Renaming logs doesn’t crash training
BT12	Training random network	Initialise a network with random weights and train it with random data, showing the network gradually converging over time	Accuracy improves over time, able to classify new data successfully
BT13	Accuracy test	Test network that has been trained for as long as possible on as much data as possible, measuring the performance calculating the percentage of successful classifications	Correct classification of 75% of classes

Live detection

Test ID	Name	Description	Expected
BL1	Default model imported	Application checks for default model store named “DOPModel.dopms. If it exists model is imported without showing error	Model imported program continues
BL2	Default model doesn’t exist	Rename or delete default model store, verify that a open file dialogue appears to pick the replacement	Open file dialogue shown
BL3	Set automatic start-up	Check that a key is created in the registry at `Software\Microsoft\Windows\CurrentVersion\Run` with the same path as the executable and id “LiveDetection” or similar	Key created, if one already exists do nothing
BL4	System tray icon	System tray icon appears on program start-up with the correct icon displayed. A menu is shown when the user right clicks on the icon and the icon is deleted on exiting the program	Icon created, menu displayed, icon deleted on exit
BL5	Show Stats	When show stats is selected a graph is displayed in the bottom right corner of the screen	Graph appears
BL6	Resize and close	Graph window can be resized, closed and re-opened without affecting the collection of classifications	Graph can be resized and closed
BL7	Classification	Use the application for 10 minutes and verify that the classification is 0, ie I don’t have Parkinson’s	Graph displays a line near 0
BL8	Enable simulation	Verify that when the simulation is enabled, the majority of classifications tend to 1 after 10 minutes of usage	Graph displays a line near 1
BL9	Physical simulation	Input mouse data at a frequency of 4-6Hz and demonstrate an increase in classification	Line grows closer to 1 with vibrations
BL10	At least (5-6)*2hz sample rate	Time the time for a new classification to be added to graph, divide this by 2048	
BL11	Resource	Start logging mouse data and while running, open task manager to verify that the Live detection application is using no more than 10% of the CPU and no more than 500MB of RAM	Uses less than 10% CPU 500MB RAM
BL12	Paint tracking	While logging and plotting is being displayed draw concentric circles in Microsoft paint ensuring smooth mouse movement, if any lines appear jagged especially when a neural network is processed if there are comparatively more artefacts it would suggest the application is affecting mouse tracking.	Paint tracks the same with/without logging

White box tests

Mouse Client

Test ID	Name	Description	Expected
WM1	Config import	Import valid and invalid config files, checking relevant variables during runtime, ensuring the URL given in the file is read correctly as are the authentication details	URL variable stores valid url, as does password and username, given authentication
WM2	Menu messages	Step through switch statement for system tray icon right click and verify that the correct functions are ran exclusively when the corresponding label is chosen	All message id's match their expected ID

Training

Test ID	Name	Description	Expected
WT1	Fourier transform	Create sine wave of frequency 5hz and verify a spike at 5 of the output. Achieved by adding a code snippet and logging the result to the console. Repeat for superimposed 7Hz + 90Hz + 450Hz	Spike at index 5, spike at indexes 7,90 and 450
WT2	Logs	Change log path to one with repeated file names in various directories, files with differing extensions and a known number of valid files.	Same number of files detected, all file paths are correct
WT3	Matrices multiply	Test small matrix operations on the CPU and verify using online tools, then verify for GPU and larger matrices. Achieved with the use of http://www.bluebit.gr/matrix-calculator/matrix_multiplication.aspx And randomly filling matrices	GPU product same as CPU and online
WT4	Small network train	Create a small network of topology 2,4,4,1 and train with xor gate input data, run in diagnostic mode and verify that relevant weights are increasing/decreasing	Weights change as expected, error decreases
WT5	Fourier Transform Iterations	Create an iterator before starting the Fourier transform method, and for each operation iterate its value and read the total for various sizes	N log n iterations

Live detection

Test ID	Name	Description	Expected
WL1	Graph	Trace a number of classifications from sample collection to plotting	NN output is expected, as shown on graph
WL2	Parkinson's simulation	Plot normalised data against the same normalised data with parkinsonian characteristics simulated	Normal graph is smooth, simulated graph undulates with sharp peaks
WL3	Watch mouse refresh rate	Place a watch (using visual studio) on the interval variable when live data is prepared and verify that this interval is within 33% the documented sample rate interval of the mouse analysed 33% Is chosen as this is the percentage of the range of frequencies to detect	Equal to roughly the refresh rate of the mouse in use

Technical solution

Overview

My technical solution is composed of three parts that are run to completion in sequence.
The first part of the solution is the Mouse Client this is a client that autonomously collects mouse data and uploads it to a web server for storage.
The second part of the solution is the Trainer it takes all collected mouse data and constructs a neural network to recognise mouse data showing the symptoms of Parkinson's.
Finally, is Live detection this is another client that runs on a user's machine collecting mouse data and using it to classify them on a spectrum of parkinsonian behaviour.

A more detailed look

Mouse client

Sets up auto starting with windows, registers mouse devices for logging, creates a window to read messages, dispatches mouse movement messages, storing and uploading them incrementally to a server.
Configurable with a config file and controlled with a system tray context menu.

Training

Deep searches for log files within a given directory, based on user decision creates a new random network or imports one from a model store binary file. Create 1000 classes using mouse data, 500 showing no signs of Parkinson's, 500 showing parkinsonian like behaviour apply normalisation and pass through Fourier transform in preparation for training. Choose a random starting point in the array of "Null" (non-parkinsonian) and "Target" (Parkinsonian like) data. Iterate through training data alternating between null and target data. Adjust weights after running a forward pass with the input data and back-propagating any errors. With the ability to do so on the CPU or GPU, this section has been targeted for the GPU as it offers greater performance, reducing lengthy training times.
Give a status report testing on the training data every 100 iterations.
Backup the model every 100 iterations
Every iteration check for user input with the key combination Ctrl + Shift + O signalling the user wishing to export the current model.
After each iteration push a new class to the queue of classes popping the last one from the queue, alternating the added class between null and target.

Live detection

A client that runs like the mouse client collecting mouse logs in the same way, but uses a trained neural network imported from a default file (if the default file cannot be found the user is prompted to locate a model store) to classify the user every 2048 mouse samples using the output of the neural network after the input has been normalised and passed through a Fourier transform. When the user wishes the results are plotted on an OpenGL graph, made from primitive types (lines and text), that can be resized for easier viewing.

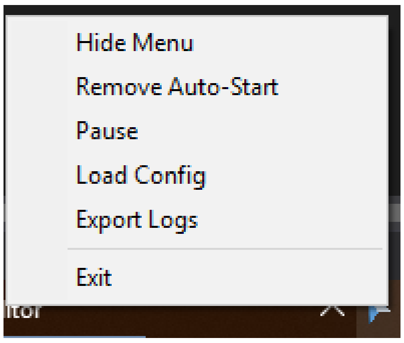
Note: all text in **bold** is a reference to the code dump, the path given in bold represents the structure of headings and titles that the code can be located in.

Mouse client

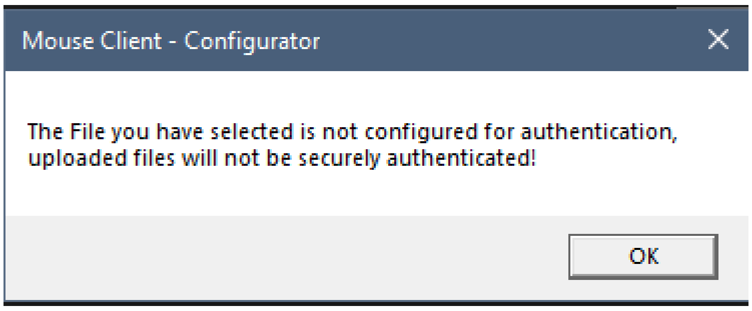
The mouse client does not differ to my design by a large extent, I added a hide menu option to the system tray menu as there was no message signalling the user clicking off the menu so this was the easiest solution to allow the user to cancel picking an option. You can see the setup for the menu in **Mouse client/Main** in the Win Main function (the entry point for the program).

Normal operation

Under normal operation the program begins logging mouse data and displays a system tray icon with a corresponding context menu.



This is the context menu shown to the user when they right click on the system tray icon, it lists all the options available to the user. There is very little user interface for the program as the intention is for the program to run in the back-ground without the user having to be aware of it at all times. All other output is through message boxes like shown bellow or though renaming the system tray icon (shown later).

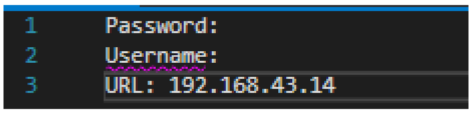


Mouse Client - Configurator

The File you have selected is not configured for authentication, uploaded files will not be securely authenticated!

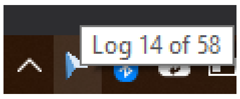
OK

Here is an example of a standard message box, the example shown is displayed when there isn't any authentication details in the configuration file, the configuration file to give this output is:



```
Password:
Username:
URL: 192.168.43.14
```

Output is displayed through the system tray icon when logs are being exported to update the user with the progress of the export. An example of this export is shown below:



I have structured the code with 3 main functions, they are:

- WndProc: this is the call back that's passed whenever a message is received and is a default definition for windows 32. I declare the function before instantiating it as it is referenced in CreateWinAPIWindow (discussed later) since the WndProc also calls CreateWinAPIWindow so one of the functions had to be defined before instantiated. It simply switches between each case of message I have chosen to respond to.
- Create win API window: Registers a window class and creates a widow at the default location
- Win main: Sets up the main program class, MyWMInput (this is described later). Creates the main message window followed by closing the widnow to hide it from the user, register mouse devices to the new window. Setup the system tray menu and its icon, the icon is loaded from the "SystemTrayIcon.ico" file in the executables directory. Followed by processing messages until the program exits.

Mouse client class

To store variables across each function I declare a global class which encapsulates the methods and properties I need in the message loop since I cannot directly pass the function to the loop as a parameter. **Referencing Mouse Client/Lib,** this stores the time deltas for sample logging, the structures required for the retrieval of mouse data, the binary to csv converter and the array of menu items, all of which must be configured in the main loop but used in the message loop.

When the application is called the setup function for this class is called. This is the method that configures all the properties of the class finding the path to the application and configuring auto starting, reading the configuration file for the FTP server authentication information, setting up the first sample log and recording the time.

This class also holds the method that is called when a WM_INPUT message is received, you can see that in the WMInputHandler I retrieve the mouse data by requesting mouse data from the operating system using the handle referenced by the lParam of the message and cast this structure to the RAWINPUT structure which gives me access to the last x and y deltas. I also retrieve the time then write a sample to the log.

Next, I check if uploading is enabled, if so I check the number of message iterations, if this is greater than or equal to 8192 (8*2048) I attempt to upload the sample log to the server referenced by the configuration file.

Sample logging

Sample logging is handled by my binary storage class it has 3 methods, the first is an open file method, this is filled with error checking as there could be a number of errors that occur when opening a file, the chief among which is the application having insufficient permissions to write to the file. To check for these issues, I look for a change in the last system error when creating the logging directory (if the directory already exists nothing will happen) then when opening the recording info file.

The recording info file is a simple text file in the recordings folder which stores the number of recordings allowing the application to increment a file ID according to the number of logs in the folder, this helps prevent overwriting a sample log. To ensure that I don't overwrite a sample log, before opening the log I check if the file exists, if it does, I increment the file id and try again, if the file exists the stat while be 0. Before opening the file, I also write the file id to the recording's information file for later reference. Finally, I open the log and check if the log is actually open, if not return an error.

The next method is Close File, this simply closes which ever log file is currently open.

Finally is write sample, this method takes the three arguments that are the data I wish to log to the file, being the X and Y coordinate deltas and the corresponding change in the time stamp.

Then I simply write them to the log with their defined lengths (4 bytes for a long, 8 for a 64bit). The data types are implicitly converted to characters as this is the only data type that the standard libraries file streamer can handle but means each variable must be treated as an array of 4 or 8 bytes and written byte by byte to the file.

Binary to CSV

This is the method used to export logs to excel, its main method takes a pointer to the storage class currently in use. The reference to which is used to close the current log file and determine where logs are stored. If a log is currently in use I set a flag and store its current write pointer such that writing can be continued without impairment afterwards.

I then create a new file to store the CSV excel data requesting the user for the path in the process (as long as the path given by the user wasn't null), using a "Get save path" defined earlier.

Before opening the file I ensure the path ends in .csv.

Once a valid CSV file is open I then open the logs using the same recursive log searcher used during training, I have described this later in training – log importing. This function simply finds all log files and writes their paths to an array and returns the number of files in that array. I then loop through each path in the array and open the log. I calculate the number of samples in the log and read each sample followed by writing it to the CSV file, I also give a status update by changing the name of the system tray icon as logs are exported. Once all logs have been exported to the CSV file I close the file and check if there was a binary file being written to previously, if so I re-open that file and reset its pointer to its previous value. I also reset the system tray icon name to mouse client.

Configuration manager

Referencing Mouse client / configuration manager

The configuration manager serves to retrieve the server configuration and control how and when files are uploaded to the ftp server. When the application starts the password, username and URL of the server are imported as described in my design, if the application cannot locate the configuration file the user will be requested for the path which will be set using the set config file function. Finally when a file is uploaded the upload file over ftp function is called. This functions by calling a precompiled "FTP_EXEC.exe" program which takes an array of strings being the username, password and URL of the server, the program is documented later. I attempt to upload the server 5 times before disabling uploading.

FTP Upload executable

Since delving into the C++ socket library was out of scope of my project, I decided to use Microsoft's .NET framework to upload files to an FTP server.

Referencing Mouse Client / FTP executable

This is a simple script using standard code from MSDN to upload files to an FTP server. The main function takes an array of strings from the command line as arguments (when it is called in my project using WinExec it can be likened to creating a virtual console) if there are 2 arguments given for example "SampleLog.dopbf ftp://DOPFTPServer.com" the second is assumed to be the URL of the FTP server which does not have authentication so the file will be uploaded without authentication. If there are 4 arguments for example "SampleLog.dopbf ftp://DOPFTPServer.com admin password" it is assumed the file is to be uploaded with authentication and the program acts accordingly.

Training

Training the network is carried out through a console which loads logs from a directory and training an existing or new network, iterating through each log to provide data to train with, simulating parkinsonian behaviour every other iteration to train the network to recognise it. Every 100 training iterations through every training class in the queue (100000 backpropagations on single classes) the output and the target output for every class is printed to the console.

Data Preparation

Fourier Transform

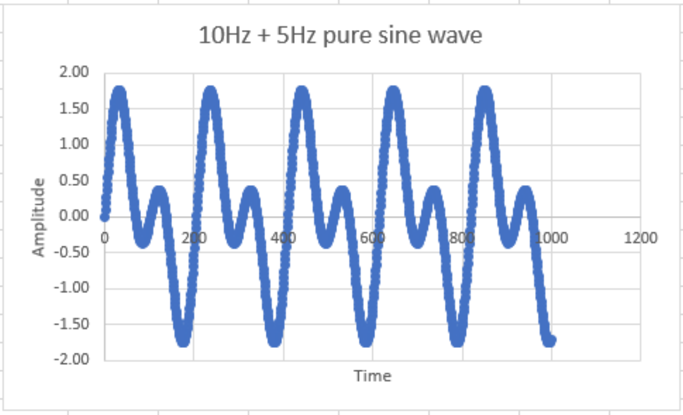
Referencing Training/FFT

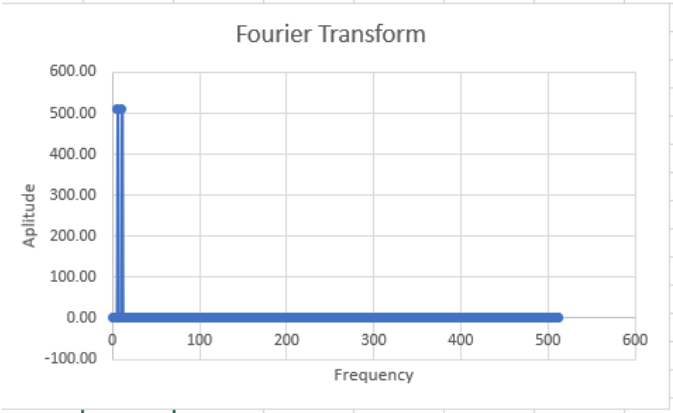
The FFT function is implemented in its own header, it contains the “Split odd index’s” function, the main “Fast Fouierer Transform” function and a utility function: “AddSine”.
I first declare Pi to an extreme number of decimal places, all references to Pi in the main function will take this value. It has been defined with the “#define” statement so it is a global constant availible for the compler to replace with the given value in every reference.
The Fast Fourier transform is a recursive function with a base case when the size of the given array is less than two. In the normal case the first call is to “SplitOddIndexes”.

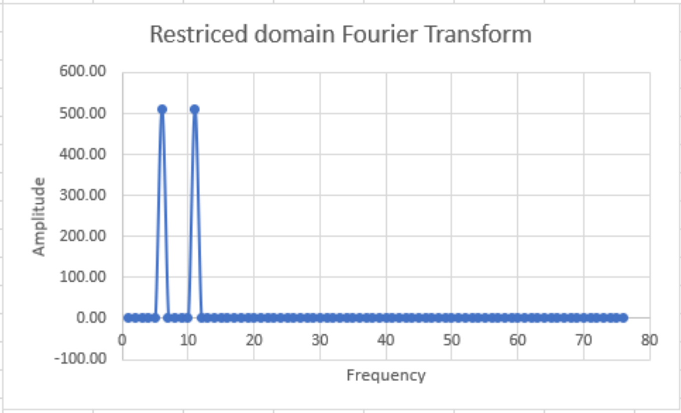
The split odd indexes function, as the name suggests, splits all the odd indexed elements in the array such that even index’s are in the first half of ComplexArray and all even are in the second half of the array. This is achieved by creating a temporary array for the odd indexes, named “Temporary Odd Buffer” it is a pointer to an array of complex numbers. The memory for which is allocated with the malloc function, the malloc function allocates a number of bytes in series in memory, and returns a void pointer. To calculate the size of the allocation required, I multiply the desired size of the array (Half the size of the input array) by the size of a single complex double precision float structure. Since the function returns a void memory address pointer I reinterpret its value as a complex double pointer to store it in the Temporary Odd Buffer variable. Now I have a suitable array, I copy each odd index from the input array to the temporary array, using a for loop with half size iterations. I then using another for loop with half size iterations, shift all the even indexed values to the first half of the array. Next I copy from the temporary buffer to the second half of the input array. Thus the input array now has even indexed items in the first n/2 indexes, and previously odd indexed times in the second n/2 indexes.
Finally the pointer to the “Temporary Odd Buffer” array is freed and the memory for that allocation is freed.

After splitting the indexes, the recursive call is made, Fast fouier transform is called for the first half of the split array, and then for the second. This in turn triggers a stack of calls each processing the first half of their given input until the input size is less than two. Once the base case of every lowest level recursion has been reached the function then beigns to work its way back out of the stack. As each call exits it loops though the first half of the given array where each iteration has index “k”, in each iteration adding $(\cos(2k\pi) + isin(-2k\pi)) * k^{th}\ even\ indexed\ element$ to that kth element then subtract $((\cos(2k\pi) + isin(-2k\pi)) * k^{th}\ odd\ indexed\ element)$ from that odd indexed element each for the kth element up to that call of the FFT. Then the function exits that layer of recursion stepping up to the next layer of recursion.

For early testing, I the generate 2 pure sine waves of frequencies 5 and 10Hz and superpose them, then input them into the FFT function and print the contents of the array to the console so I can copy them to an excel spreadsheet for viewing. The results of which are shown below.


10Hz + 5Hz pure sine wave
Amplitude
Time
2.00
1.50
1.00
0.50
0.00
-0.50
-1.00
-1.50
-2.00
0
200
400
600
800
1000
1200


Fourier Transform
Aplitude
Frequency
600.00
500.00
400.00
300.00
200.00
100.00
0.00
-100.00
0
100
200
300
400
500
600


Restriced domain Fourier Transform
Amplitude
Frequency
600.00
500.00
400.00
300.00
200.00
100.00
0.00
-100.00
0
10
20
30
40
50
60
70
80

Matrix Library

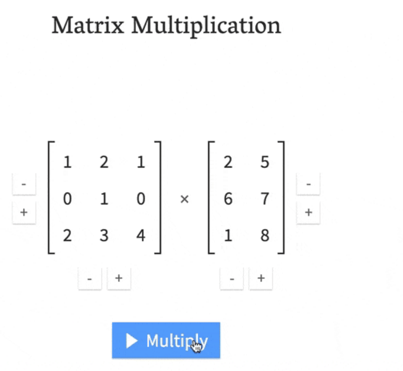
Class

Referencing the code dump **Training/Matrix Class:CPUMatrix** you can see the class encapsulating the structures and methods used for storing a matrix on the processor. The key properties are its dimensions and the array of double precision floats, the x and y sizes must be stored as they define the matrix dimensions, I have used the “size_t” type to store these dimensions allowing the program to decide at runtime which variable size to instantiate for maximum efficiency. Size is stored to reduce the number of calculations during training as, although it can be computed easily as the product of the x and y dimensions, the number of element wise operations conducted during training that only care about the size of the array and therefore the size of the matrix, warrants the extra storage rather than having to retrieve 2 properties and compute their product. I could access the same value through the vector size method but this is a member function of the vector subclass so will take longer to access.
I chose a singular dimensional array to store the network in due to the classes transpose ability, in which the x and y dimensions swap, if the array was 2 dimensional it would have to be redeclared with new fist and second dimension sizes increasing the processing time to compute the transpose.

Main functions

Referencing the code dump **Training/Matrix: Matrix maths namespace** there are a number of functions used during training.

CPU Multiplication

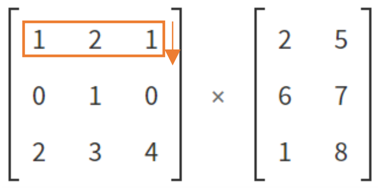


The single most used method in the project, matrix multiplication multiplies two matrices and stores the result in a new matrix. Given two matrices of size R1 by C1 and R2 by C2 the method will produce an output matrix of size R1 by C2 however due to the manner in which matrices are multiplied, see the GIF (right), the columns C1 and rows R2 dimensions must be equal. You may think of a matrix multiplication as the mapping of one matrix onto another with the rotation of the second 90 degrees left then performing the dot product of each combination of rows.

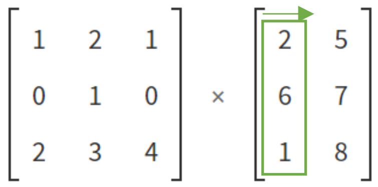
Throughout this explanation I refer to the x and y members of each input matrix class A and B as R and C as members of matrices 1 and 2, as this matches the notation used for the iterators in the for loops.

Before execution I check that the R1 and C2 dimensions of the input matrices are not different, otherwise I throw an error.
I then construct a new return variable C the product matrix into which the result of the multiplication will be stored. Consequently the dimensions of this return matrix are equal to the number of rows in matrix, A, followed by the number of columns in the second matrix, B.

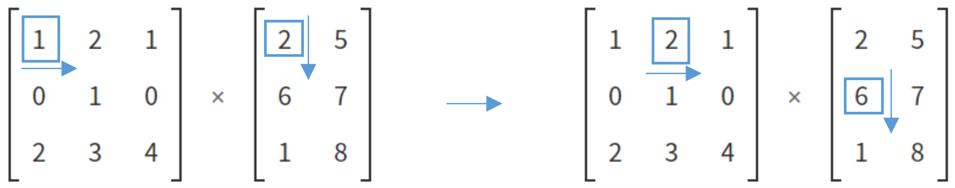
The method achieves the multiplication through the use of 2 nested for loops within a main loop that iterates through each row of the first matrix: R1, in this case that's thrice.



It then iterates through each column C2 of the second matrix, in this case that's twice:



Finally we iterate through each column of the first matrix C1 or if you prefer each row of the second matrix R2, since these dimensions will be equal, in this case that's 3 iterations



We then multiply these two elements and store them in the product matrix at index R1, C2 in this case 0,0. Note this means that all multiplications carried out in within the same iteration of the second loop, will be added to the same index of the product matrix.
Due to the appendage of values to each index of the product matrix, it is critical that the product matrix contains only 0's. Accordingly I fill the product matrix with 0's before performing the operations.
I have chosen to use the size_t variable to store the dimensions to allow for the executing machine to decide on the most efficient integer size to use.

Transpose multiplication

The CPU transpose methods: “CPU Multiply A”, “CPU Multiply B”, “CPU Multiply AB” in the matrix maths namespace serve to **transpose** a given parameter A, B or both A and B.
Since the computation of a transpose of a matrix is simply the act of switching the rows and columns of the matrix, the effect can be achieved by switching the indexing of the Matrix arrays such that it indexes by column then row rather than by row then column. This is achieved by simply switching the index parameters. Since the transpose switches the dimensions of the matrix all references to the transposed matrix must be switched.
If we take CPU Multiply A as an example you can see that all references to the dimensions or index of A have been switched. In the first line, checking for equal inner dimensions. You can see that I am now comparing the number of rows in a rather than the number of columns in one.
Likewise in the product matrix definition and the 3 for loops. Finally you can see that the A.Index statement has had its parameters switched.
In the following example you can see how switching the index’s parameters essentially looks up the transposed matrices indexes.

If we consider A as:

$$\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{pmatrix}$$

$= [1,2,3,4,5,6,7,8,9]$

And consequently its transpose:

$$\begin{pmatrix}1 & 4 & 7\\ 2 & 5 & 8\\ 3 & 6 & 9\end{pmatrix}$$

$= [1,4,7,2,5,8,3,6,9]$

Since the index function is: $index(Y, X)\{\ Y * xdimension + X\ \}$ (x-dimension = 3)
If we input X as the Y parameter the result will be the beginning of the row at that index rather than the corresponding column, then adding the Y as the X parameter gives us the corresponding column.
If we test this out and attempt to index 6 in both arrays we can see that passing (1,2), since we index from (0,0), the function will return 5, if we lookup the 6th index we see that it is 6 in the array for matrix A, however for the transposed array the value would be 8.
If we now swap the indexing and pass (2,1) to the index function, it will return 7, if we now lookup the 8th index in the transposed array we can see that it is in fact 6! Thus by switching the index parameters we can effectively transpose the matrix without any prior computation.

CPU-Add

This method computes the element wise addition of two matrices, it is used when applying biases to the un-activated output of each layer during the neural network main process. And simply iterates through each index of each matrix adding the contents at the two indexes together. Thus a single for loop is needed for this iteration. Since this function will never be applied directly to user input it is assumed that the sizes of the two matrices are equal to save computation time. The result is stored in a new matrix of equal dimensions and returned.
In example the addition is as follows.

$$\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{pmatrix} + \begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{pmatrix} = \begin{pmatrix}2 & 4 & 6\\ 8 & 10 & 12\\ 14 & 16 & 18\end{pmatrix}$$

CPU-Add constant

This method computes the addition of a single value to every element in the matrix, the item to add is given as a double precision floating point integer “B”. A resultant array is constructed with the same dimensions as the given matrix “A”. In a single for loop which iterates through each element of the matrix the value is added and stored in the resultant matrix C. C is then returned. This method is used during training to compute the optimised equivalent function discussed later to the sigmoid function Outputs(1 - Outputs).

Sigmoid

Computes the sigmoid of every element in a given matrix with the sigmoid transformation being: $\frac{1}{1+e^{x}}$
Since this is applied element wise a single for loop is needed iterating up to the size of the network.

CPU Scale

Multiplies each element in a matrix by a single double precision floating point value, this is used primarily during training to scale the error signal using the learning rate to decrease the change in the weights.
A single for loop is used to achieve this iterating through each element of the matrix upto the last element. The resulting values are stored in a new array of equal size as the input and returned.

Element wise CPU

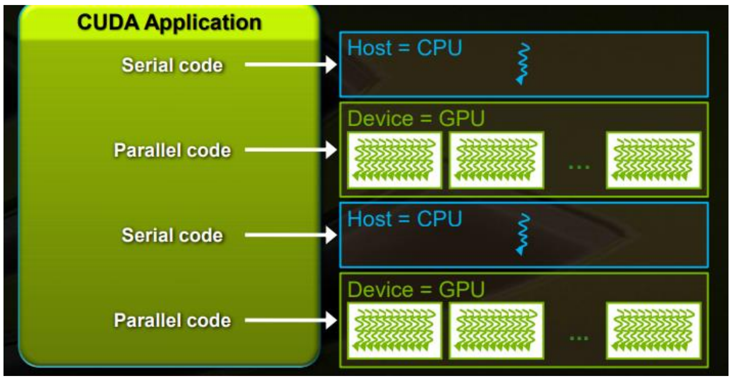
Multiplies to matrices together element wise, this means multiplying each element of the matrices together in turn given that the two input matrices are of equal dimensions. The result is stored in a new matrix of equal dimensions.

GPU acceleration

I will be using CUDA compute to accelerate the training of my network since the GPU has much greater facility to multiply matrices – the biggest task undertaken when training a network. Thus I have extended the matrix library to contain matrices stored on the GPU device and repeated all CPU matrix math functions for the GPU.
To understand my code you must first understand how matrices are multiplied on the CPU here is the method I wrote for CPU matrix multiplication which I use during normal execution as the vast majority of the target audience will not have dedicated graphics with CUDA compute capability, especially since a high spec card is required to make the GPU compute worth it.

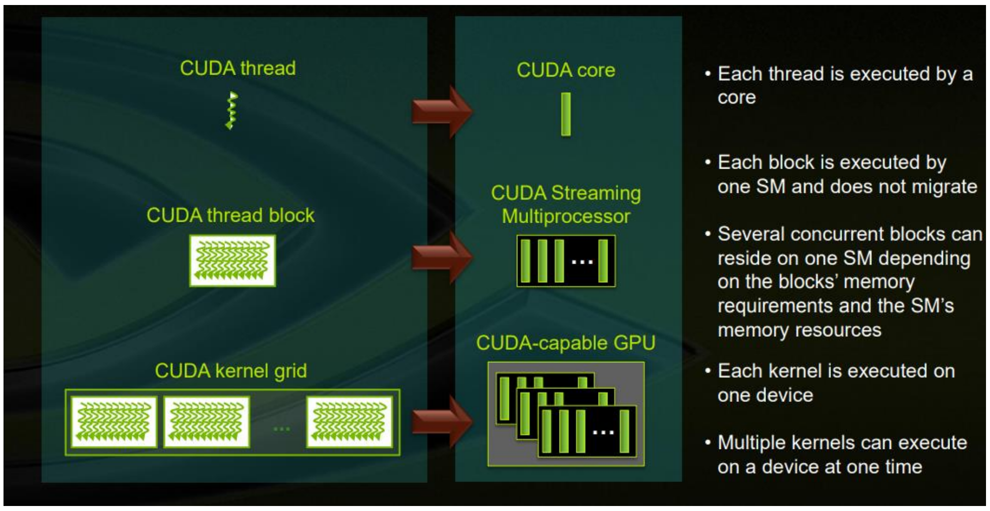
CUDA

CUDA is a technology developed by nvidia to accelerate compute workloads. It functions by using stream units on discrete graphics cards which, although limited in flexibility can perform double precision calculations at much greater speed than the processor, not due to increased instruction execution but due to the highly parallelised nature of how they compute. All graphics cards have many stream processors each of which can be thought of as a stripped down CPU core, Nvidia brands these stream processors as CUDA cores.
The first method I wrote was to find the number of stream processors that can have an open thread on the device. For me this is 1024 as I have a 780ti.
Each thread on the GPU can then run functions in parallel with every other thread on the GPU. This allows for highly parallelised work loads to be completed much quicker. For multiple operations to be completed on a single thread as if they were parallel you must then employ the use of blocks. Each block runs on a single thread and each block has dimensions (up to three) every index in every dimension is then ran in serial. With the GPU managing the order of execution. This allows for operations with a complexity greater than 1024 to also run as quickly as possible.
In the diagram below you can see how the host (CPU) executes a single function, which then triggers a cascade of threads to run in parallel, each of which with a block ran in series. The white squares each represent a thread. The squiggles, each index in the block, being ran sequentially within the thread. On completion the CPU then only has to execute a small amount of code in serial before the vast majority of computation can be handled by the GPU.



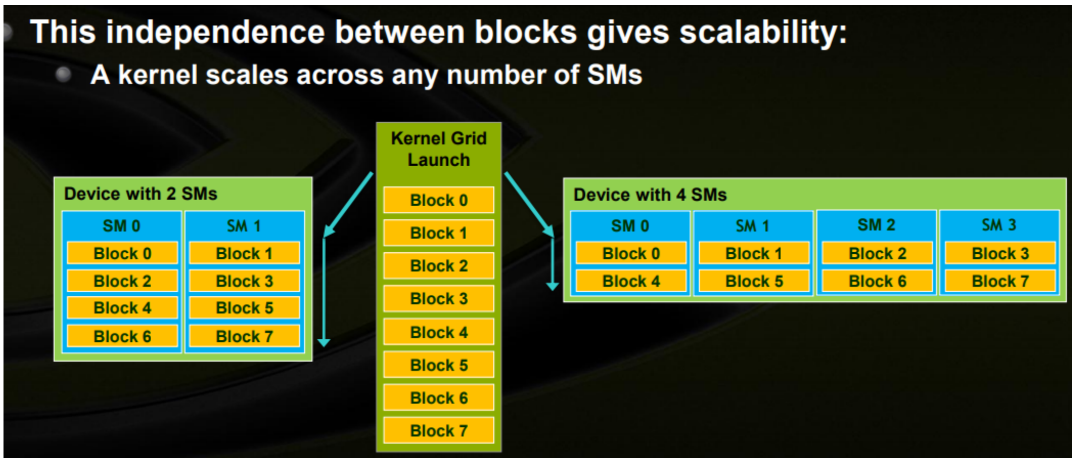
Each block executes what is known as a kernel a function intended to run on the GPU instantiated with the dimensions of the threads and the blocks. The kernel executes low level code that has access to the index of the thread, and the index(s) of the block within that thread, it can also be given extra parameters such as pointers to the start of arrays on the device.

Bellow you can see how each thread can be split into many blocks which can then each run a kernel:



- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

Below can also see how, if the number of stream processors vary, and therefore the number of possible threads varies the number of blocks can change on each thread to compensate for the lack of threads.
In the example below, 8 operations are to be run in parallel, for a device with 4 threads each can run just 2 of the 8 operations. If the device only has 2 threads each thread must then execute 4 operations and thus the kernel will take twice as long to execute on the second device.



- **This independence between blocks gives scalability:**
  - **A kernel scales across any number of SMs**

It is also worth noting the issues that arise due to sub-dividing an algorithm to run on a GPU since they are well accustomed to algorithms with a 2^n complexity that can be divided many times without any blocks having nothing to run. I describe later the approach I have had to take to counter this issue.

GPU Matrix Class

Referencing the code dump **Training/Matrix: Matrix class (GPU)**

The GPU matrix class exits to encapsulate the storage of a matrix on a GPU, the key issue with this is that there is no direct memory access from the CPU to the GPU. Consequently none of the advanced data structures used previously can be used such as std::vector, as they assume the array is to be stored on the host. Instead the array is stored sequentially following a pointer to the start of the array. The double precision floating point data type pointer is the same as that used on the CPU but when allocated the referenced memory location will not be accessible by the program as it instead references a location on GPU memory. Since the only variable we are storing relating to the matrix array is the pointer to the start of the array, the class must crucially store the size if the allocation as there is no other way of knowing. Although the size, x and y variables existed for the CPU matrix they are critical for the GPU matrix, otherwise the device memory allocation could be easily overflown. Since we which to dynamically allocate the matrices on the GPU rather than have the sizes static, I have employed the use of “cudaMalloc” this allocates a given number of bytes on the GPU’s memory and stores the pointer to the start of the allocation in the first parameter, the pointer to our array. Since double precision floats use 8 bytes of storage, I use the “sizeof” operator to calculate the size of the allocation necessary.

In summary, on construction the matrix class will allocate the required amount of memory on the device and store the pointer to this allocation in the array pointer, additionally the dimensions and size of the array will be stored within the class. The size and dimensions of the matrix are stored on the host (CPU) since they are never directly used during matrix maths.

Since the array is not stored in host memory, if it is needed by the host the contents of that index in device memory must be copied to the host. I use the “cudaMemcpy” function to achieve this which is given the pointers to the target location, in this case a memory address on the host; and pointers to the memory address of the source, in this case an address on the device. The function finally requires the type of operation to be carried out, in this case a copy from the device to the host.

If a value is needed to be copied from the host to the device the converse is carried out, where the source is a location in host memory, the target is a location in device memory and the type of operation is host to device.

With the above foundations laid the rest of the class is very similar to that of the CPU matrix class, just with the appendage of the new memory access methods. The main other difference is the added complexity in the Random Fill method used to instantiate weights and biases. Since the first layer weight matrix has a large number of elements: 787200. To store every single one with separate CUDA memory copy statements is extremely inefficient. Consequently, I allocate an array of equal size on the host, fill it with random values then copy the entire array to the device at once. Followed by freeing up the array.

GPU Matrix maths

As I have mentioned every function in the CPU matrix mathematical library has been duplicated to run on the GPU. Since the vast majority of latency is due to the interaction between the CPU and GPU I have modified the parameters of the majority of methods to accept the output matrix as a parameter, rather than constructing a new matrix after every operation. Additionally, memory management on the GPU must be carried out manually and is not controlled by the operating system to the same extent as system memory. Consequently, I have helped mitigate the issue allowing all the matrices needed during training to be constructed at the start of the program such that no new matrices need to be created.

GPU Matrix multiplication

The CUDA kernel contains the code that will be ran directly on the GPU, it has been logged in **Training/Cuda Code.**
The code in the GPU kernel relies on the thread ID and block ID to identify the instances position in the call. They represent effectively each index in each nested loop in the matrix multiplication CPU function.
In order to have the kernel execute correctly you must declare the dimensions of the kernel before it is called this can be seen in the **Matrix Math** namespace in the GPU multiply functions.
In order to keep the GPU kernel calls not to dis-similar to that
To execute matrix multiplication on the GPU device I first check the dimensions are valid as per usual.
Next I determine the dimensions of the threads and blocks to execute the multiplication. For my implementation I have considered two options to maximise the similarities between the GPU Matrix multiplication and CPU matrix multiplication. Both of which map the 2 for loops to the thread/block dimensions.
Side note: only the first two loops in the triple loop structure in the CPU multiply code can be ran on the GPU this is due to the last loop appending to the previous value of a single index in memory, if all three components of the loop where mapped to the GPU, they would all be ran in parallel, this would cause all of the final loops to attempt to write to the same memory location at the same time, which is not possible, consequently on one of the block index's would be allowed to write, causing the wrong resultant value to be stored in that index as it would only be the result of a single multiplication rather than a row of multiplications.
The first method is named "Thread mode". This is called if the first two loops, if the multiplication was running on the CPU, have a product less than or equal to the number of threads available on the GPU. This maximised the number of open threads and therefore the number of parallel operations occurring each second, reducing the time taken for the operation which is simply then the sum of the serial operations in the final loop of multiplication. In this case I set the thread x dimension equal to the number of rows in matrix A, this matches the first loop in the CPU multiplication. I then set the thread y dimension equal to the number of columns in matrix B, this corresponds to the number of iterations in the second loop of the CPU equivalent operation. The last loop in the for loop is ran sequentially within the kernel. The reasons for which are described in the side note above. In summary "Threaded mode" runs the first two loops of the multiplication in parallel, and the last in series.
The second is "non-threaded mode" this occurs if the product of the size of the first two loops is greater than the number of available threads. Consequently we can only run the first loop in parallel, on its own thread, to implement this I set the x dimension of the thread structure to the y dimension of Matrix A; this is the same as in thread mode so represents the number of iterations in the first for loop in the CPU operation. All other thread dimensions are one. The second loop then must be ran sequentially so the block x dimension is set to the number of iterations in the second CPU loop, being the x dimension of matrix B. in summary this "non-threaded mode" runs the first loop of the matrix multiplication in parallel, while all others are ran in series.
Due to the nature of CUDA processing the GPU device will automatically assign some blocks to an individual thread if not specified thus performance is still nearly optimal even if some threads are not in use.

If I was to redesign the GPU acceleration I would research methods of more optimally distributing the load across different blocks and threads, to ensure the GPU is under full load at all times. As it is during training at least 70% of the GPU is constantly in use, I think that is an ok level as there is some processing time between GPU operations to account for the lower utilization. Additionally there is a limitation on the maximum size of the matrix multiplication, as depending on the GPU device the matrix is limited to the number of threads available to perform the operation. This is not an issue in my scenario since the number of threads possible on my graphics card is 1024, equal to the largest possible matrix multiplication with network dimensions 1024,768,512,256,1.

Once the dimensions have been declared for the kernel. The matrix multiplication kernel is called with CUDA kernel parameters denoted in the <<< >>> operators. Here I pass the dimensions of the blocks and threads. I then pass the pointers to the matrix array. The referenced memory allocations are resized before calling any multiplicative methods so memory overflow does not have to be considered. I have extensively tested various sizes and have not experienced any. Finally I pass the x dimensions of all passed matrices, this information is required for the indexing in the array as the arrays are indexed by rows then columns (y * X + x). The A matrices X dimension serves a second purpose as it is the number of iterations in the final loop located within the kernel.
Speaking of the kernel, the application will now enter the corresponding “threaded/non threaded” kernel. The kernel will be called as many times as the product of all blocks multiplied by the product of all threads. Each individual kernel will be called effectively knowing which iteration in the second loop it is in. For threaded mode this will be given by the thread x index as the index in the first loop, for the second loop it’s the thread y index. Additionally for the non-threaded mode first loop’s index will be given by the thread x index but the second loop is given by the block x index. The kernel then executes the dot product of the given row in A (given by the thread x index in both cases) and the given column in B (given by the thread y index or the block x index for thread and non-threaded mode respectively). Since the kernel is called in parallel every combination of row-column dot products are performed and the matrix multiplication is complete!
Note the index function is referencing the force-line index function, which as the name suggests forces the index function: “`y * XD + x`” when it is referenced, filling the given parameters.

The functions have been repeated for all possible transpose configurations where multiplication A represents the case when matrix A is to be transposed, multiplication B is where B Is to be transposed and finally multiplication AB is where both matrices A and be are transposed. All these transposed variants implement is the switch of all index and dimension operations carried out on the given matrix. This is the same process as carried out for CPU multiplication, only there are far more dimensional references.

GPU sigmoid

The first declared function in the header is tagged as “force in line” this means that they must be referenced within another CUDA kernel. The first example of this is GPU Sigmoid, it simply takes an input parameter as a double and returns the result of the sigmoid function also as a double. This function is used when applying the sigmoid function to an entire matrix. The kernel for which, conveniently is declared after.
The GPU Sigmoid Array kernel applies the aforementioned sigmoid function to an entire array, since the matrix may be viewed as simply a vector, or single dimensional array, there are standard methods for applying CUDA acceleration for a “map” function, in this case we are mapping the input matrix’s vector array through a sigmoid function, meaning the function is applied element wise.
In essence the kernel need only be called x*y times, consequently there are many indexing methods. The one I have used caps the number of threads to 512 since all CUDA compute 2.0 graphics cards (v2.0 is required for double precision calculations) have at least this number of threads. Next the dimensions of the block is determined as the remainder of the iterations required that cannot be for filled by a thread. In the kernel l itself any remaining index’s are handled within each kernel call.

Element wise multiply

Element wise multiply functions the same as element wise multiply on the CPU, multiplying every index of the first matrix by the corresponding matrix in the second. It uses the simpler but less efficient method I used for the GPU matrix multiplication as it is far more readable and the element wise multiply takes negligible time in comparison to an exponent function. The function employs two threading modes one if all iterations can be assigned its own thread, another in the case where the second loop has to be ran sequentially in a block. Once each iteration has been assigned a thread id or block id the kernel is called and the corresponding indexes are multiplied and stored in matrix C.

Add constant

Uses the exact same indexing strategy as element wise multiply but calls a kernel that adds a constant to the single input matrix.

CUBLAS based operations

For some functions that I had insufficient time to manually implement I have used the CUBLAS compute API from Nvidia, this handles array/vector operations and is extremely well optimised. The functions utilizing the CUBLAS API as follows:

Create CUBLAS handle

All CUBLAS API function calls require a handle to be referenced, this handle stores the specification of the GPU so that the library can fully utilise the available device(s). The create CUBLAS handle function is called at the start of a script, for convenience I added it to the constructor of the GPU neural network as it is a necessity for its operation. The create CUBLAS function first instantiates a Status variable and sets its value to failure. This Status value is then assigned a value when the CUBLAS API “cublasCreate” function is called given the null pointer to a handle structure passed to the function. The API call then returns a status message, if this message is anything but success the method will loop until the creation is successful. When the handle is successfully made the method will then return the “EXIT_SUCCESS” flag.

Scale

Firstly for scaling (multiplying every index in the matrix by a constant) a copy must be made of the input matrix, this is due to the CUBLAS function applying the scale to the input parameter, if a copy was not made the input would be modified. I first create a CUDA error structure and assign it launch failure, I then call the copy until the operation is successful. The cudaMemcpy function takes the destination pointer, the source pointer, the size of the array in bytes and the copy type as parameters. Next a similar tactic is employed for the GPU scaling as was used when creating the handle, a status variable is initialised as a failure, then the CUBLAS operation is carried out until it return success, if there is a failure I also recreate the handle as I found during testing that this almost always resolved any errors. The CUBLAS API function used here is cublasDscal, it scales an array of given size (number of doubles rather than bytes) with a pointer to the constant scalar and the output array. The final parameter is another constant “step size” if you only wish to scale every n^{th} element.

Add

The add function adds two matrices of equal size together and stores the result in the given matrix “C”. this requires two operations first, like scaling is the copying of the, in this case, second input matrix to the output, otherwise its contents will be modified. Then we call the cublasDaxpy function given the input matrix and the output matrix C, where C is equivalent now to B. each element in A is then added to each element in C and the result is stored in C. The function parameters are the CUBLAS handle, the size of the matrices, all are assumed the same, a pointer to a scalar to be applied to matrix A (this is set to 1 for no scaling), the pointer to the array to add (matrix A), the step size for the first array, then a pointer to the array to add to and store in, finally another step size for the second array. Like before I repeat the operation until it is successful, recreating the CUBLAS handle after each failed attempt.

Machine learning

The machine learning in the solution is handled with two distinct elements, the first is the CPU network and training utilities, these allow for training without a graphics card at the cost of a decreased training speed for large networks. The second part is a GPU network and its corresponding training utilities, all matrix data is stored in device (GPU) memory for quick retrieval rated at 7gbps for my older 780Ti, with a relatively low latency, for new cards using HBM memory this increases to 307GB/s!
There are three main stages to training. The first is locating and importing logs to memory. The second is simulating parkinsonian behaviour on existing, known healthy mouse data. And finally is the network training itself.

CPU Network

Referencing Training/Network/CPU

Neural Network class

The neural network class encapsulates all the data required during training and execution of a neural network of given dimensions. The default properties are a number of arrays of matrices, these arrays are vectors so can be dynamically resized during runtime. The matrix arrays and their role during training are described below: (The number of arrays could be reduced but an increased number greatly aids debugging and the detail obtainable when training in diagnostic mode, described later)
Each index in the vector indicates the layer to which the matrix applies to.

- Outputs: Stores the outputs of each layer in the network starting at index 1, index 0 is actually the inputs for the network as this makes the feed forward and back propagating methods far easier to follow, for the default network topology this gives output sizes of (1024,1), (768,1) …. The second dimension is always one to indicate a single row as the input and output should be a vector.
- Weights: stores the weights for the network, these are large matrices used to store the weight of each neuron connecting two nodes. They each will have the y dimension equal to the size of the output vector for the layer and x dimension equal to the size of the input vector for that layer, in example the first matrix has dimensions (768, 1024).
- Biases: These are the biases added to the output of each layer before the activation function (sigmoid) has been applied. They, as suggested by the fact the must be added to the outputs, have the exact same dimensions as the outputs for which ever layer, they are applied to.
- Raw Outputs: Used during diagnostic feed forward processing to store the un-activated outputs before the biases have been applied, concequently they have the same dimensions as biases and therefore the output for the layer they apply to.
- Gradient: stores the gradient of the output of each layer, this is used after calculating the error signals and is the sigmoid differential of the output element wise multiplied by the error for that layer, thus scaling the sigmoid gradient of the output by that node's responsibility for the error at the output. Due to this element wise multiplication, the gradient must have the same dimensions as the output for that layer.
- Error signal: stores the error for at each output with respect to the overall output of the network, calculated by back propagating errors through the network. Since the error is associated with the output of a layer all error signals have the same dimensions as the output of their corresponding layer.
- Weights D: the change in the weights to be applied after each training iteration, this is the result of multiplying the gradient the output for that layer by the transpose of outputs from the previous layer (otherwise known as the inputs for that layer), this calculation results in a matrix with y dimension equal to the outputs for that layer and x dimension equal to the inputs to that layer, thus it has the same dimensions as the weights for that layer, so the two matrices can be added together.

Construction

The network is constructed with a default constructor, in C++ the default constructor is declared when a member function has the same name as the parent class, in my case this is Network CPU. I have added a null constructor and passed over construction to an external setup function to allow the network to be declared outside a of a main loop, this allows me to declare the structure without constructing it so I can reference the variable in global functions without needed to pass the network as a parameter, this is required for the windows message loop as its default parameters of HWND (window handle),message, WPARAM and LPARAM, do not allow for the addition of a CPU Neural network parameter. Consequently, the network had to be declared globally in live detection, warranting the null constructor. The non-null default constructor takes a parameter "Dimensions" this is a vector, of any size that contains a series of numbers, the number of numbers i.e. the length of the array indicates the number of layers in the network. The value of each layer represents the size of the output for that layer with the first representing the size of the input.
The dimensions are then passed over to the setup function which constructs the corresponding weights, biases... in each layer to the correct size, the size of each is described above with reasoning.
I also store the number of layers for the network in the NoLayers variable.

Neural network process

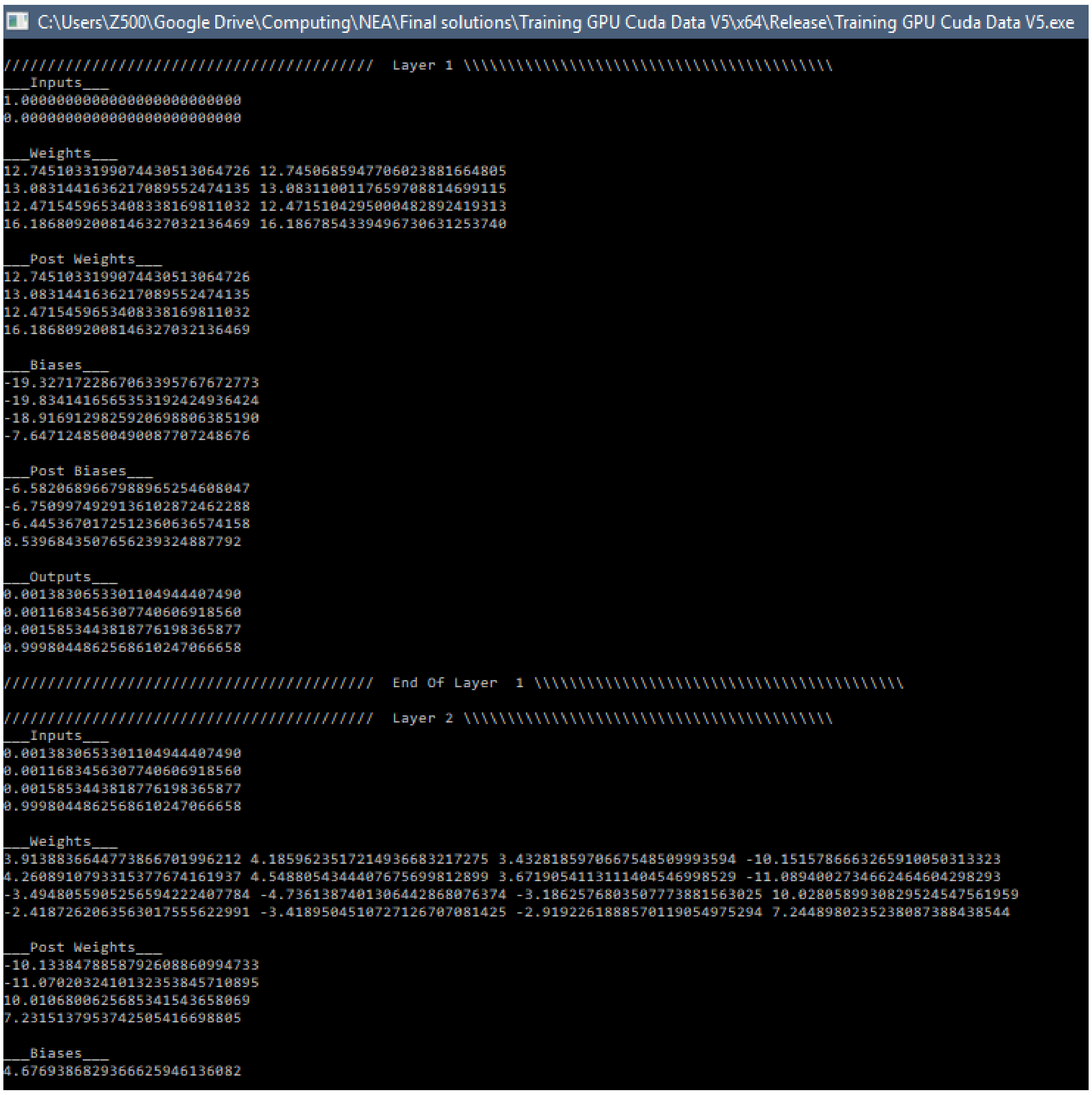
In this function I run the forward pass for the neural network, this consists of iterating through each layer of the network, multiplying the input to that layer by the weights, applying the bias to the product matrix then applying the activation function to the result, until the final layer is reached.
This is achieved by starting a for loop from layer zero (the index of the inputs in the array of output matrices) to the last layer, one index of the output array from the final output matrix.
In each iteration I multiply the weights in the next index (the indexing for weights starts at 1 to match the layer number) by the input to that layer. This results in a matrix of dimensions (Weights. y, outputs. x = 1) note this gives the a returned matrix of size equal to the inputs to the next layer. I now apply the biases by adding them to the result of the "weight, input product" using another matrix math function. The result of this is then passed through the sigmoid function. Which normalises the output of that layer to values between 0 and 1.
Each layer if the trained network will serve to recognise contrast in the data, an example of this is in the first layer the final index of the input matrix is the sampling interval of the data, the trained network should use this value to increase the value of the index in the input that corresponds to frequency responses in the range 4-6Hz which highlights any contrast between these indexes and the rest of the input, thus highlighting a known component of Parkinson's, the trained network will be able to use much more information and will classify based on a complex function applied to the inputs capable of recognising subtleties in the input data.

There are a couple more basic variants of the neural network, the next takes a vector array of doubles, since the vector array structure also stores the size of the array, the size does not need to passed. This variant uses the input parameter as the first layer of the network, consequently I create a new layer 0 matrix using the inputted array, since it should have the same array dimensions as the Outputs[0] array. Next I run the first layer of the feed forward network outside of the main for loop so I can use the custom input layer matrix for the first iteration, I then enter the main for loop used in normal feed forward execution, starting from the next layer (2nd). This method is used when data has been generated without the use of a Fourier transform, giving the library scope to test with this component disabled.

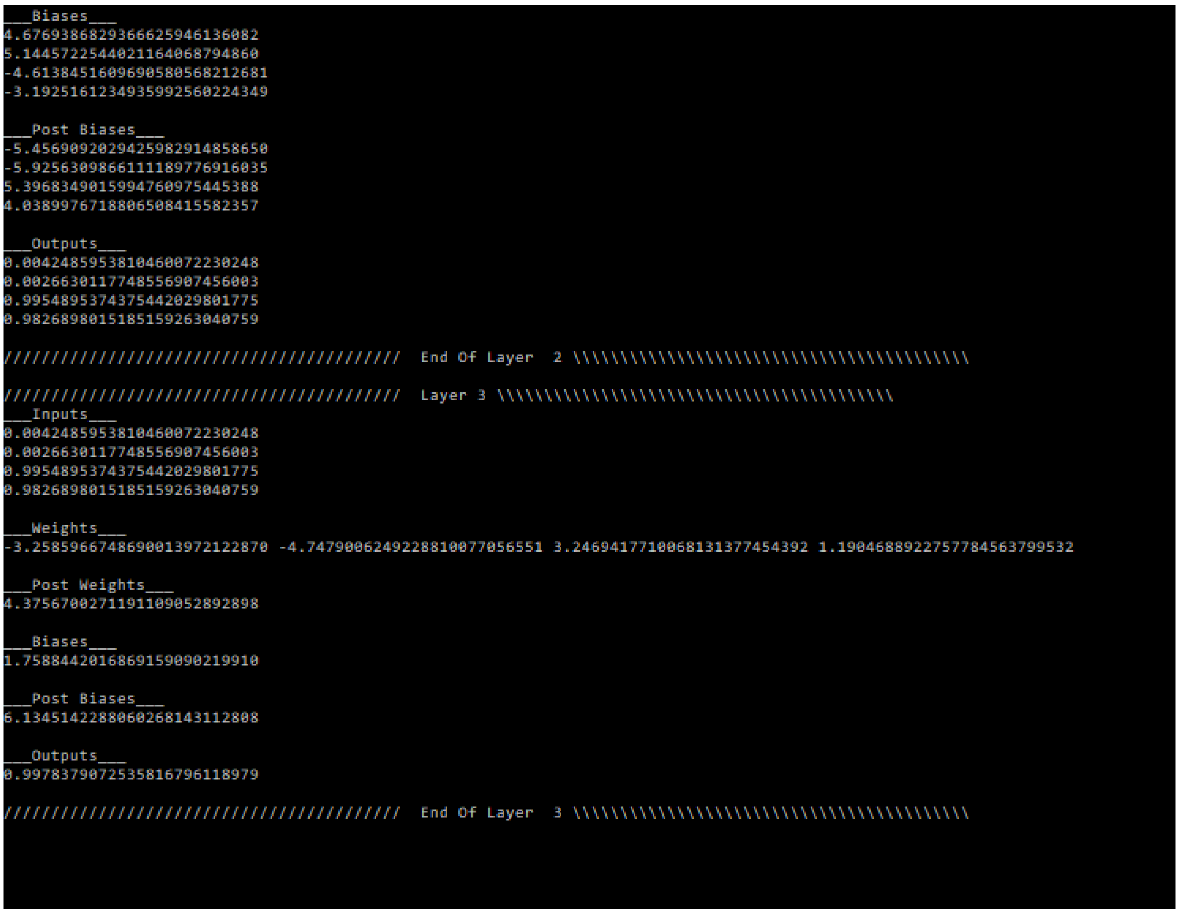
The next variant is very similar to the prior but uses a static array of complex numbers, indexed from the start with the pointer passed in the first parameter, since this pointer holds no other information the size of the array must also be passed. However since the static array is not a vector type and contains complex numbers we cannot directly assign it to a new matrix due to this I added a matrix multiply function that takes the B input as a static complex array of a given size, this restricts the x dimension of the product array to 1. The matrix multiply function simply uses the absolute values of each index in the complex array to perform the calculation. Like before this first layer is processed outside the main loop, but on completion enters the loop.

Finally I have repeated the neural network processing functions with diagnostics, this has been used to produce the forward pass with full diagnostics described later. In essence it breaks up every operation and stores the output, printing them to the console with the relevant identifiers. The function first prints the current layer being processed with a long string of slashes to make it obvious where each layer starts and finishes. I then print the inputs to the network followed by the weights they will be multiplied by. I then perform the multiplication of weights by inputs and store the results in a temporary "post weights" array. This post weights array is then printed to the console with the corresponding identifier. The next step of the forward pass is to apply the biases. Accordingly I print the biases, perform the addition, store the result and display the result, with the apt identifier "post biases". Finally I apply the sigmoid function and store the result as per usual execution and display the output of the layer.
All matrix prints are printed in columns then rows. The values are delimited by spaces so that they can be read by my checking tool: a website that allows for the easy multiplication of matrices online using text copied from the console, helping the verification of the neural network process.

Demonstration on feed forward pass, using trained XOR network:



```
C:\Users\Z500\Google Drive\Computing\NEA\Final solutions\Training GPU Cuda Data V5\x64\Release\Training GPU Cuda Data V5.exe

////////////////////////////////////////  Layer 1 \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
___Inputs___
1.000000000000000000000000000
0.000000000000000000000000000

___Weights___
12.745103319907443051306472б 12.7450685947706023881664805
13.0831441636217089552474135 13.0831100117659708814699115
12.4715459653408338169811032 12.4715104295000482892419313
16.1868092008146327032136469 16.1867854339496730631253740

___Post Weights___
12.745103319907443051306472б
13.0831441636217089552474135
12.4715459653408338169811032
16.1868092008146327032136469

___Biases___
-19.3271722867063395767672773
-19.8341416565353192424936424
-18.9169129825920698806385190
-7.6471248500490087707248676

___Post Biases___
-6.5820689667988965254608047
-6.7509974929136102872462288
-6.4453670172512360636574158
8.5396843507656239324887792

___Outputs___
0.0013830653301104944407490
0.0011683456307740606918560
0.0015853443818776198365877
0.9998044862568610247066658

////////////////////////////////////////  End Of Layer  1 \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

////////////////////////////////////////  Layer 2 \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
___Inputs___
0.0013830653301104944407490
0.0011683456307740606918560
0.0015853443818776198365877
0.9998044862568610247066658

___Weights___
3.9138836644773866701996212 4.1859623517214936683217275 3.4328185970667548509993594 -10.1515786663265910050313323
4.2608910793315377674161937 4.5488054344407675699812899 3.6719054113111404546998529 -11.0894002734662464604298293
-3.4948055905256594222407784 -4.7361387401306442868076374 -3.1862576803507773881563025 10.0280589930829524547561959
-2.4187262063563017555622991 -3.4189504510727126707081425 -2.9192261888570119054975294 7.2448980235238087388438544

___Post Weights___
-10.1338478858792608860994733
-11.0702032410132353845710895
10.0106800625685341543658069
7.2315137953742505416698805

___Biases___
4.6769386829366625946136082
```



```
___Biases___
4.67693868293666259461360822
5.14457225440211640687948600
-4.61384516096905805682126810
-3.19251612349359925602243490

___Post Biases___
-5.4569092029425982914858650
-5.9256309866111189776916035
5.3968349015994760975445388
4.0389976718806508415582357

___Outputs___
0.0042485953810460072230248
0.0026630117748556907456003
0.9954895374375442029801775
0.9826898015185159263040759

//////////////////////////////////////////  End Of Layer  2 \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

//////////////////////////////////////////  Layer 3 \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
___Inputs___
0.0042485953810460072230248
0.0026630117748556907456003
0.9954895374375442029801775
0.9826898015185159263040759

___Weights___
-3.2585966748690013972122870 -4.7479006249228810077056551 3.2469417710068131377454392 1.1904688922757784563799532

___Post Weights___
4.3756700271191109052892898

___Biases___
1.7588442016869159090219910

___Post Biases___
6.1345142288060268143112808

___Outputs___
0.9978379072535816796118979

//////////////////////////////////////////  End Of Layer  3 \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
```

You can see above each stage of executing a 3 layer neural network with dimensions 2,4,4,1. Each layer is shown in full until the output is reached. The desired XOR and actual outputs of the network are as follows:

Input	Target	Actual
00	0	0.0022142013878784744078387
01	1	0.9978379072535816796118979
10	1	0.9978379073152788825140647
11	0	0.0019977283784136231833961

Training

Training the network is the most mathematically complex section of my project, here the weights and biases are adjusted until the outputs converge to a desired value. In my project this is handled with three functions all called in series when the main backpropagation function is called. These are, calculate error signals, update weights and update biases.
During development I found an optimisation when updating weights which is documented below.

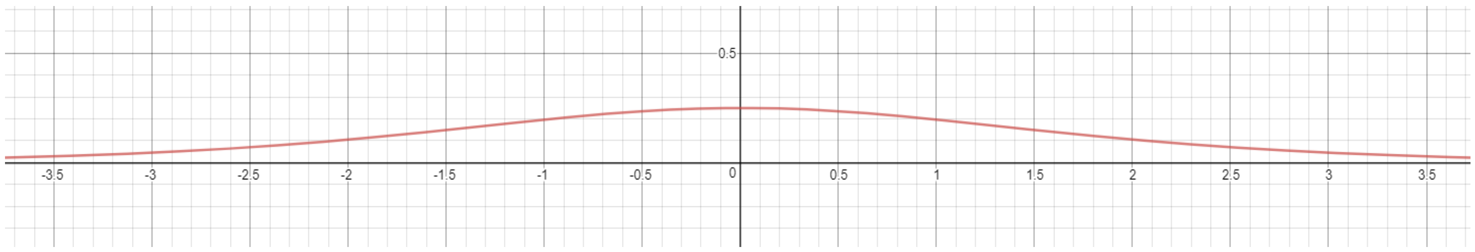
Optimisation

As I have coded my solution, I have realise that there is a decision to be made with regards to the backpropagation of errors since the gradient defined below obtained with the chain rule does have an alternative.

$$\frac{\partial a_k^l}{\partial z_k^l} = \sigma'(z_k^l)$$

This gradient serves to reduce the change of the weights as the raw output of a neurone reaches negative infinity and infinity. The equation is derived from the application of the chain rule to obtain a result for the gradient of a weight with respect to the error function which is then used with the delta rule to update the weights in the network. This has the effect of preventing the weights from growing to extreme values.
If we plot the function here, $\sigma'(x)$.

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



As you can see output of the differential of the sigmoid reaches 0 as the input nears infinity and negative infinity. In terms of the network this means that as the input of the neurone nears these extremes the weight will change less. Also you may notice that the equation involves the calculation of e^-x which is a computationally difficult function and thus requires comparatively long time to execute than e raised to some integer value. Since backpropagation will require the sigmoid differential to be called for every weight in the network of which there are 1283328 it makes sense to try our best to optimise these equations.

We also have the result that $a = \sigma\left(z_k^l\right)$ thus there is a relationship between the activated output and the sigmoid differential applied to the raw output. This relationship is that as z nears infinity and negative infinity the activated output nears 1 and 0 respectively. So as the activated output nears 0 and 1 the change in the gradient should near 0.
We can then see that an alternative equation that would behave similarly is that

$$\frac{\partial a_k^l}{\partial z_k^l} \approx a(1-a)$$

As the input nears 0 and 1 the output of our new function nears 0 thus it seems like a good replacement for the sigmoid differential.

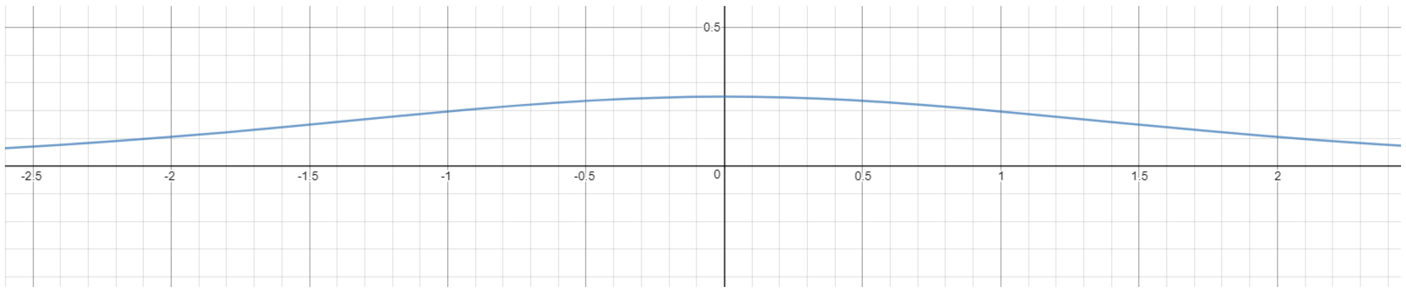
We must now decide whether to stay true to the definition derived through calculus which will is more precise but highly time complex or to use the far simpler and less computationally complex $a(1 - a)$.

To achieve this, we must write both in terms of the same variable and compare the fit of the graphs.
We can use our result from earlier to help us with this: $a = \sigma(z_k^l)$, we can then substitute $\sigma(z_k^l)$ for a into our new function to be tested, this results in:

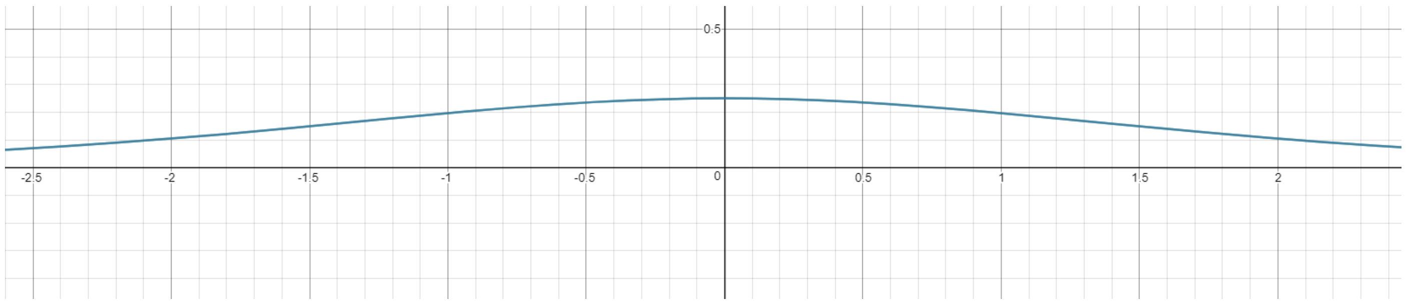
$$\frac{\partial a_k^l}{\partial z_k^l} = \sigma'(z_k^l) \approx \sigma(z_k^l) * (1 - \sigma(z_k^l))$$

$$\frac{\partial a_k^l}{\partial z_k^l} = \frac{e^{-x}}{(1+e^{-x})^2} \approx \frac{1}{1+e^{-x}} * (1 - \frac{1}{1+e^{-x}})$$

After plotting both we se that $y = \sigma(z_k^l) * (1 - \sigma(z_k^l))$ as z_k^l varies is



And when plotted on top of each other:



We can now see that the two functions are in fact identical! Thus, no precision is lost when making this optimisation. Consequently, my backpropagation methods follow this in order to greatly decrease training times. We can now write:

$$\frac{\partial a_k^l}{\partial z_k^l} = \sigma'(z_k^l) = a(1-a)$$

Calculating error signals

Error signals are calculated by backpropagating errors through the network. This is well documented in my design. In the technical solution I have implemented the algorithm with two for loops. The first for loop iterates through each output of the network, calculating the associated error, the second for loop propagates this error back through the network in line with my design. With the exception of the multiplication involving the transpose of the weights instead, otherwise the dimensions are invalid for multiplication.

Update weights

The update weights method differs from that in my design as I have integrated the scaling of the error signals into the main loop. Additionally, I have applied the more computationally efficient method of calculating the gradient for which the sigmoid differential is equal to the output element wise multiplied by 1 – the output. This is achieved by scaling the outputs by minus one then adding the constant 1 to all indexes then multiplying the resultant sum by the outputs. The function then continues as per the design where Outputs[layer]. transpose is replaced by CPU multiply B where the B parameter Outputs[layer-1] (the minus one is due to the shift due to the input layer) is transposed.

Update biases

This method is very simple and has not changed since designing it.

Train

Combines the neural network process followed by calculating the error signals from that iteration, then using these error signals to update the weights and biases adjusting the network to better classify in future.

GPU network

Referencing Training/Network/GPU

Neural network (GPU) class

The neural network class for operation on the GPU is very similar to that for the CPU, however it features more matrix members. The reason for which is that the GPU takes a comparatively long time to allocate memory when compared to the CPU equivalent, thus it would lead to inefficiencies during training if the application was wasting time allocating device memory. Instead I allocate all matrices for the given network size before training and pass both the input matrices and the matrix to store the output in so that no new matrices have to be constructed. Since the GPU network is intended for maximum training performance, I have sacrificed the memory requirements for performance.

The majority of the matrices are the sane as in the CPU variant, with the exception of the following:

- Scaled sig stores the error signals after they have been scaled by the learning rate.
- One min outputs stores the sum of the negation of the outputs and the constant 1, this is used during training when calculating the change to the weights
- Output differential stores the product of one min outputs and the outputs matrix as this forms the sigmoid differential of the outputs
- Post weights used during the feed forward process, it stores the product of the weights and inputs to that layer with biases added, in other words the output of each layer before the sigmoid has been applied.
- Weights copy is used when updating the weights as the addition function does not add to an input matrix, instead it stores the output in a different sum matrix, consequently the weights are copied to weights copy and then weights copy is added to weights deltas and the result is stored in the usual weights matrix.
- Biases copy is defined as the same process has to be carried out for biases too.

I also have to store the CUBLAS handle, a handle to a structure that stored GPU information, I describe this earlier in my technical solution. Likewise I store the max number of threads for the given GPU device rather than retrieving this before each matrix kernel is executed.

Construction

Construction is nearly identical to that for a CPU targeted neural network; the construction has the same null constructor and the normal constructor calls a similar setup function, the only difference in this setup function is that the CUBLAS compute is constructed, populating the structure with device data. I next check how many threads are on the device and assign the “Max Threads” variable to this value.

Randomise weights and biases

This function triggered the alteration in the GPU matrix method to populate itself with random values in a given range as I found that randomly filling the GPU matrix when the application was started took a considerably long time as the CPU was retrieving and storing 1310976 weights and 2560 biases from the GPU device. This increased the time to generate the network massively. As a result I create an equivalently sized weights matrix on the CPU using the malloc memory allocator for main system memory , and fill this with random values. I then copy the contents of this CPU array to the GPU once it has been populated using the CUDA memory copy function. Finally I free the CPU array from memory.

Main process

The main process is not too different to that for the CPU, each operation is carried out separately since the matrix functions don’t return the resultant matrix. Each layer is iterated through, the weights are multiplied by the inputs for that layer and the result is stored in post weights, the biases are added to the post weights and the sum is stored in the raw outputs. Finally the sigmoid function is applied to every element in the result is stored in the outputs array which will be used as the inputs to the next layer.

Calculate error signal

Calculating the error signals using the GPU, backpropagating errors is identical to the CPU version, I have chosen to keep the error signal calculation for the last layer (first iteration) on the CPU as there is only one output from my network, it also gives me added flexibly to change the error function with out having to redesign a CUDA kernel.
The backpropagation by multiplying the transpose of the weights by the error for the next layer is the same as the CPU but is executed using a GPU matrix maths function as the weights matrix are large so the product would take a long time to calculate on the CPU.
There is also a diagnostic mode which makes use of the extra matrix declarations.

Updating weights

Is more complicated on the GPU as the weights cannot simply be appended to, instead a copy must be made and the copy must be used to add the weights to allowing the weights deltas to be applied.
The other change is again due to the output being stored in a pre-constructed matrix passed to the function. Consequently the update weights function is far more complex.
Again there is a diagnostic mode making use of the extra matrix definitions, printing them to the console.

Updating biases

When updating biases I make another sum to the biases; this requires that I make a copy of the biases to append the values to, so that I can store the result in the actual bias array, similar to how I append values to the weights.

Train

The train function is the culmination of the neural network process followed by backpropagation (calculating error signals and updating weights and biases accordingly) followed by calculating the error in the final output.
I also check here for infinities which can arise if an error occurs during training, if it does the training iteration is skipped.

Log importing

Logs are stored in “dopbf” (detection of Parkinson’s binary file), these store a sequence of samples and are described in my design. Training will use a multiply log files located within a directory. In order to find and store the location of the logs I perform a recursive search for “dopbf” extensions and store the path to them in an array of strings. **Referencing Training/Training classes,** the recursive search is located in the training class which stores an array of classes for training. In the classes constructor I pass a path to a folder. The path to this folder is found by requesting the user to use file explorer to select directory that contains logs. The functions to request file paths / directories can be found in **Training/Training files**. Going back to Training classes, the search within the directory is recursive such that it can navigate directories within the given directory and search for logs in there, this makes it far easier to import logs as the user doesn’t need to move all logs to the same path. The call Is made to my search algorithm within the constructor to the open log method located within the binary storage class (in training files), this takes the directories path and applies a directory iterator, although the iterator is listed under experimental methods it has been confirmed to be stable since C++ 17 for use with windows. This iterator is passed to a for loop, in each iteration of this for loop I can access the iterators properties which contains information concerning the current file. If the extension to the current file is “dopbf” I add the full path to it to an array of strings (resizing said array prior), otherwise I check if there is no extension for the file at all, if so, it is a directory so I call the function again given this path beginning a recursion into that directory. If neither extensions are found the file is ignored. once every folder in the directory has been iterated over, I return the number of logs I have found.

Adding classes

Once I have found an array of paths to log files, I construct a queue of 1000 null and target logs to train with at any given time. I use the circular training data class **referencing Training/Training utilities.** This is a simple queue that stores the inputs for a class and its corresponding outputs. To create this array in the main trainer I use the training classes “Add class” method, this alternates between adding null and target classes to the queue, incrementing the size of the queue until its size is the desired size of 1000. I cannot simply iterate 1000 times as errors often occur when adding a class such that the size of the queue is not incremented. The two things that signal failure are the binary log file the data is read from being less than 2048 samples long or an error occurring in the Fourier transform due to an extreme value, this can result in an infinity in the result of the transform which I check for and return failure if necessary.
During training after every training iteration (one sweep of training in the entire queue of 1000 classes) I pop the last item from the queue, generate a new class and add it to the queue, making sure I alternate between null and target classes.

Simulating parkinsonian behaviour

Parkinsonian behaviour is simulated with the use of two functions, one is the add sine function and the other is add cog wheel, I document these later in the formatting section of my documentation, but when they are used for training I had to make a decision for the parameters.
I set all the parameters for the sine function to random within given ranges, the frequency is between 4 and 6 hz as this was found in my research to by typical for tremors from people with Parkinson’s. Next is the amplitude, this is again random between 0.2 and 1, I have given a large range as I believe the amplitude is the largest factor that makes the disease easier to diagnose, in turn the maximum amplitude is equal to that of the variance. The X offset is configured to ensure the network doesn’t recognise only a single phase of sine wave so is in the range 0 to pi to give potentially a full shift of phase. Finally, is the y offset, this is again to make the network more flexible and shifts the sign wave up or down a bit, as I don’t wish to change the mean too much this is configured as -0.2 to 0.2.
Next is the add cog wheel motion, this function automatically randomises the values given a parameter, how it does so is explained in formatting, I chose the maximum amplitude to be just 0.3 so that the effect is rather subtle, the duration lasts a maximum of 30 samples so the motion lasts ~300ms, and finally the occurrence is up to 40% of the time.

Formatting

Referencing Formatting

In this chapter I explain anything I missed when designing the formatting headers, in detail, explaining the rationale behind; normalisation, average interval calculation, filling gaps in the samples and adding parkinsonian characterising's with sign waves and a cog wheel motion. But first I hope to explain how random numbers are generated in my project.

Random number generation

In my projects I use two random number generators, both use the same method for generation, the first is random integer generator, this generates an integer in the desired range, the second is a random double which generates a double precision floating point integer in the desired range. I created a class for this generation as creating the random number generating classes takes a considerable time, when they can be reused, this increases performance as random numbers are used throughout my project, and if there was a long delay for each retrieval performance would drastically decrease.

Both variants have the same members constructed with different parameters. The first is a random device this is a uniformly distributed, pseudo random non-deterministic number generator, if a random number generating hardware device is not given the same sequence can be produced by this generator. I then declare mt representing the mt19937 class, which applies the Mersenne twister, a pseudo random 32 bit number generator, the fact that the twister is only 32 bit is ok as the range of values Is never large the resolution is still acceptable. Finally I have a uniform real distribution declared as RandomDist, this uses the given Mersenne twister to retrieve a random number in the given range, it is constructed with a type parameter of double to indicate a uniform distribution of double precision floating point numbers.

When the class is constructed I initialise all the properties by instantiating the Mersenne twister with a seed given by the random device followed by discarding the first few thousand iterations of the random device based on the current time in micro seconds, this counter the issue described earlier where the random device can generate the same sequence of numbers, giving the Mersenne twister the same seed. Finally I construct the random distribution for the given range using the Mersenne twister.

To retrieve a random number from the class I simply request the next random number from the stack from the RandomDist variable.

The same strategy is employed for the random integer distribution but the RandomDist variable is constructed given the data type lass rather than the double data type.

Normalisation

Normalisation has barely changed in its functionality since my design, although during development I decided to add some error checking to ensure the size of the array requested to be normalised is greater than 0, this is due to the mean being calculated by dividing the final x term by the size, if the size is 0 the application may crash. Additionally, I assign the new samples time and time delta properties as they are used later.

Sampling interval

This function was only briefly mentioned in my design but as the name suggests it calculates the sampling interval of the mouse in use. It does this using the assumption that the mouse has a constant refresh rate when moving as this is the case for the majority of mice, consequently the minimum values should hold this refresh rate once the anomalous results have been discarded. Some of the values may be very large as they are from when the mouse was stationary which could potentially be days! Consequently, the average is not suitable.

To calculate the refresh rate, I find the first 1/20th minimum values from the data set, since this function is applied to 2048 samples this gives 102 samples to calculate with. I find the minimums by adding ascending values to a queue once the values get to big, they fall off the end of the queue.

Once the minimums have been found I then average across the second half of these values and return this value.

Fill gaps

Fill gaps corresponds to my design for "Interpolate samples" and serves to prepare data for the Fourier transform which requires evenly spaced samples. It is algorithmically the same but I also added the case for when the gap is 10x larger than the average, as this generally represents the mouse being stationary. If I interpolated between theses values the output could give an array with almost entirely the same positions, which isn't too difficult if the mouse is stationary for just 2048x the average interval (~1ms) therefore the mouse would only ned be stationary for 2 seconds!

Get samples

Fill gaps is proceeded by a number of basic retrieval functions, these simply take an array of samples and store each X/Y or the absolute value (magnitude) of the X and Y co-ordinate of the sample to an array of doubles.

Add sign

The add sign function merely adds a sign wave to either a array vector of doubles or a static array of doubles, with a given magnitude, frequency, x offset and y offset.

Add cog wheel

The add cog wheel functions function the same as that described in my design, adding a increasingly large value to an array. There are a number of versions I use throughout the project, that take vector arrays/ static arrays and that apply the function to the input or generate a new output.
The main difference to my design is the random amplitude/duration operators and the occurrence parameter. These are applied using the given maximums where the minimums are sensibly defined, for the amplitude the minimum is simply the negation of the maximum allowing for negative "cogs". For the duration the minimum is 2 allowing the effect to slip into noise, while still being there, as I don't want it to be too easy to detect. Finally the occurrence simply directly defines what percentage of the data will be affected by the cog wheel effect, since this depends on the duration I take the average duration to be half the maximum minus 2 as you would expect this to be the mean.

Prepare live data

The prepare live data functions apply the required formatting to raw data for processing by the neural network. There are two variants: with and without simulating Parkinson's. Both apply the normalisation of the samples followed by calculating the average sampling interval then interpolating the samples. This is followed by passing the data through a Fourier transform and storing the sampling interval (scaled down to be close to 0) in the last index of the array, as this index is defined as a interval scalar which the network uses to scale the frequencies.
The second function, simulating Parkinson's, also adds a sign wave in the centre of the expected frequency range with a reasonable amplitude, then applies the cog wheel effect all before passing the data through the transform, the function then continues as without the simulation.
Now the data is ready for processing simplifying the classification in the main loop.

Live detection

Referencing live detection/ main

Live detection is the third and final section of my project it handles collecting samples and once it has a sufficient number classifying them using the neural network trained previously. It combines elements from both the mouse client and the network trainer. It also has a new element being the classification plotter, a new window that is created when requested at the system tray that displays a graph of the previous 100 classifications.

The main function is identical as that for the mouse client so does not warrant explaining, with the exception of "My WM Input" being modified such that the setup varies and the message loop is different. Additionally in the main function I setup a the neural network, this is simply done using my neural networking libraries import function which constructs a network from a given path to a model store, if the default path is not found the function will fail, if the function fails the user is requested to find a new model file and is repeatedly asked until the user exits the file explorer window in which case the program will exit. Otherwise the model is imported to memory and the main loop continues to dispatch messages. The window process is also modified since the menu is different.

When an input message is received the WMInputHandler is called (**Referencing Live Detection/Live detection header**) like before but rather than writing the sample to a binary file it is added to a sample array within memory. If the number of samples reaches the required amount the data is prepared for processing by the neural network, if the simulate Parkinson's flag has been set the data is prepared simulating parkinsonian behaviour. The neural network process is then ran using the weights and biases retrieved from the model store. The classification is then added to an array of classifications stored within the class, and a update message is sent to the graphing window, if the handle to the graphing window is null the windows 32 bit send message function will do nothing.

The other changes to the code are the user interface, this includes an option to simulate Parkinson's in the data being collected or to show a graph. If the user choses to show the results in a graph a new window with a corresponding message process is produced. The methods for the graph can be found within the plotter class. The graph uses the SFML OpenGL graphics library a simple API offering basic drawing primitives.

Plotter

The plotter for live detections handles the setup and resizing of the graph displaying classifications over time, it references code from the graphing header I created **Referencing Live detection / SFML graphing**
This header contains two classes the first of which is the line plot.

Line plot

This is a queue of vertices that each represent classifications from 0 to 1 that get drawn to the render window (an overlay on a system window for graphics). When the lien plot is created an array of vertices is created and the type for the vertex array is defined as a line strip (connected vertices) as opposed to a strip of lines (disconnected lines) or a quad…
There are two methods of adding to the line plot, the first is to add a points from an array of double precision floats each in the range 0 to 1 each of which is mapped to the correct position on both scales dependant on window size.
The second method is adding a queue this allows for the classification queue to be easily interpreted without losing its starting point.
Finally, the class has a draw method which draws the line primitive to a given render texture.

Draw grid

The draw grid class consists of a number of functions that simply divide the given window into lines horizontally and vertically, and drawing axis labels at corresponding positions, consequently there are a number of for loops in each components setup function. The grid is configured to draw 5 horizontal and vertical lines, with the verticals being labelled from 0 to 1 and the horizontals being labelled 100 to 0. The font for all text is loaded before the grid is setup such that the font is passed to the function.

Setup

Moving back to the plotter, it uses the classes defined in the graphing library to construct a graph, loading the font “Caviar Dreams” from a file in the same directory and creating the window in the correct area above the system tray that uses the call-back function passed to the constructor. In my solution this call-back function is declared later by the name handler.
Once the window has been created its window process, the aforementioned “handler” receives the size message, consequently the rest of the windows construction is also handled in the resize member of the plotter. Thus every time the graphing window is resized the graph is simply recreated (consequently the vertex arrays for which are cleared beforehand even if the resize event has only been called the once). The resize function in the plotter class receives the new size of the graphing window and accordingly attaches a new render target to it, allowing for the drawing of OpenGl elements to the window. The grid is then setup given the scaling, x and y offsets, the size of the window and the colour of each of the lines. This is followed by configuring the X and Y axis and adding the queue of samples to the line plot, finally the line plot and the grid with its axis are drawn to the screen. This approach allows the window to be seamlessly resized for easier viewing, a feature critical to assisting the user who could potentially have issues with their vision, as Parkinson sufferers are often elderly, who wish to see how the prediction changes over time.

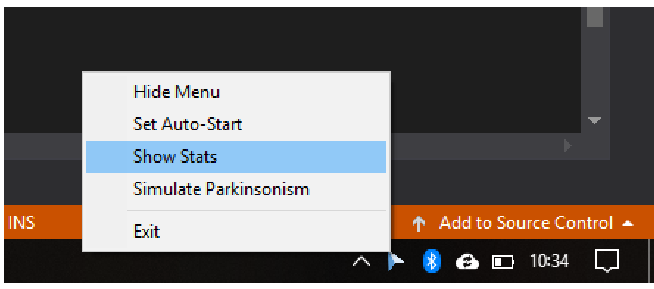
Plotter message loop

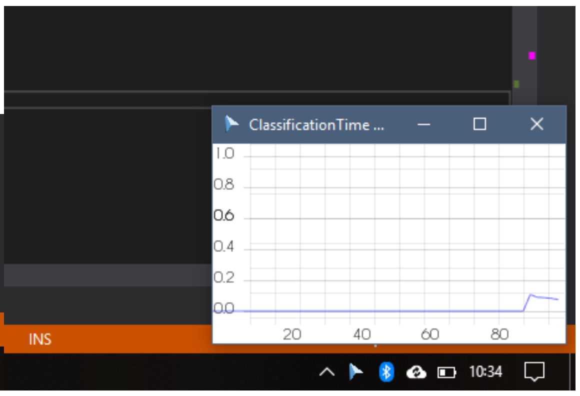
The plotter message loop not only contains the resize function described earlier it also contains the add classification message, a custom message sent to the plotter window when a classification is added to the array, accordingly the points are re-drawn from the queue to the line plot.
Additionally, there is a destroy function, this gets called when the graphing window is closed, since it may be re-opened I must clear plotter graphing classes and de-register the window so that it can be re-registered if applicable later on.

In summary a main message loop handles input messages for as long as the program is running, automatically classifying data when enough has been received. If one of the messages corresponds to the user deciding to show their stats, a new window is created and with its corresponding message loop operating in a separate thread, which handles the plotting of a graph displaying classifications over time.

Screenshots

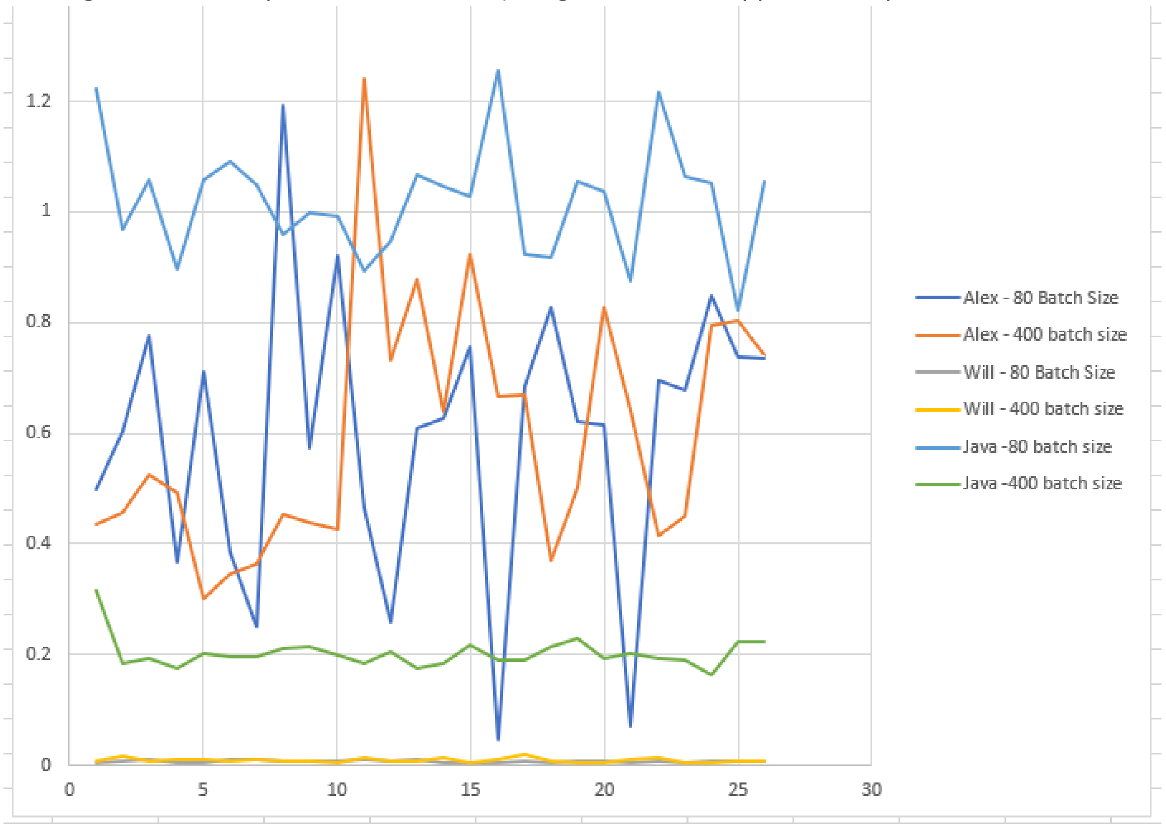
I have included a couple screen shots to show the task bar menu and the graph plotter below.





Testing

On completing my neural network and the corresponding training utilities, I tested the CPU training performance by running two tests training a very small neural network till convergence to successfully recognise an XOR function with 90% certainty. It would not be appropriate to test the GPU training on a network of this size as setup overheads would outweigh the processing time. I have compared this to training times from my friend Alex Butler (using their NEED Support Library - Alex Butler



The time in seconds is indicated in the y axis while the trial in the x axis
From the quickest (lowest) to the slowest (top) the order is as follows, Will – 400, Will – 80, java – 400, Alex – 80, Alex – 400, java – 80.

From the graph you can see that the lines marked "Will" representing the timings for my network are far below all others, calculating the averages gives that my network converges 84.86 times faster than the same network constructed in visual basic and 114.84 times faster than the same network constructed in java script.
This is due to the lower level nature of C++ and the corresponding optimisations it allows me to achieve.
The batch size of the graph represents the number of iterations before testing the network, we varied this to investigate how conducting more tests overall on the network over time affected the training time till convergence.

From my testing design I hoped to conduct the following test, I have documented each and its result below: https://www.youtube.com/playlist?list=PLoK5Ujy_qlKHaXnHcG17wrydJQ0ORhmGI

Black box tests

Mouse Client

Test ID	Name	Expected	Outcome	Testing video reference
BM1	Log creation	If Recording0.dopbf exists will create Recording1.dopbf…	Pass	https://youtu.be/x8lBNRddIQw
BM2	Set automatic start-up	Key created, if one already exists do nothing	Pass	https://youtu.be/SMaruhMesL8
BM3	Configuration file is read	Authentication disabled if user/pass fields not filled, likewise with URL	Pass	https://youtu.be/PC3N0NAsGwU
BM4	System tray icon	Icon created, menu displayed, icon deleted on exit	Pass	https://youtu.be/R299h4h0dA4
BM5	Export logs	Log.csv file is created on export csv logs. Read CSV, generate graph verify a straight line to the right	Pass	https://youtu.be/axDPQUBorWY
BM6	FTP Upload	Log uploaded to FTP server	Pass	https://youtu.be/nJCWslfKqbA
BM7	Log resolution	travel of known pixels to have greater change in log	Pass	Same video as BM5
BM8	Sample rate	Time delta across 100 samples less than 8.33 seconds	Pass	https://youtu.be/Qu6KzUsMkR0
BM9	New mouse	Logs from new mouse devices	Pass	https://youtu.be/2VTm1srJUW4
BM10	No mouse	Doesn't crash when mice disconnected	Pass	https://youtu.be/2VTm1srJUW4
BM11	Binary file	Binary file clearly shows mouse logs	Pass	https://youtu.be/Uov-5kn2lKw
BM12	Resources	Doesn't use more than 500mb of ram	Pass	https://youtu.be/g6zAqzVdICM
BM13	Compare FTP	Files sent to FTP server are the same as those on device	Pass	https://youtu.be/jx85vFkIJmU

Training

Test ID	Name	Expected	Outcome	Testing video reference
BT1	Select Log folder	On start select directory dialogue shown	Pass	https://youtu.be/oI6krKlCJQc
BT2	Exit log folder (forcefully and through close method)	Dialogue closes and program halts	Pass	https://youtu.be/3bnS4b0dM8w
BT3	Train existing - Yes	Dialogue produced if yes is given	Pass	https://youtu.be/D24Y3jckKz0
BT4	File importing normal	No error message is given, training commences	pass	https://youtu.be/LbO1kmBYh3w
BT5	File importing erroneous	Error message if the file is invalid	Pass	https://youtu.be/0ou_v1454vg
BT6	Train existing - No	Program continues	Pass	https://youtu.be/_Chmv_oa5gM
BT11	Erroneous user input	Requested to retry		
BT7	Train existing erroneous model store	Requested to retry	Pass	https://youtu.be/ikkgkNXZggw
BT8	Run time	Convergent network	Pass	https://youtu.be/688unzDkLsU
BT10	Log corruption	Renaming logs doesn't crash training	Pass	https://youtu.be/QIZNYNhwiHc
BT12	Training random network	Accuracy improves over time, able to classify new data successfully	Pass	https://youtu.be/dlnFQFGj7Sk
BT13	Accuracy test	Correct classification of 75% of classes	Pass 92.84& accuracy	https://youtu.be/S4WzKUnfQtY

Live detection

Test ID	Name	Expected	Outcome	Testing video reference
BL1	Default model imported	Model imported program continues	Pass	https://youtu.be/4gnwIw2zCFs
BL2	Default model doesn't exist	Open file dialogue shown	Pass	https://youtu.be/WgkOO6JbLCo
BL3	Set automatic start-up	Key created, if one already exists do nothing	Pass	https://youtu.be/_Zi-kdKSsPo
BL4	System tray icon	Icon created, menu displayed, icon deleted on exit	Pass	https://youtu.be/8x9rndis210
BL5	Show Stats	Graph appears	Pass	https://youtu.be/f6gSpCDDZiQ
BL6	Resize and close	Graph can be resized and closed	Pass	https://youtu.be/ggLu6iK3XBM
BL7	Classification	Graph displays a line near 0	Pass	https://youtu.be/DSB3EQVq3F0
BL8	Enable simulation	Graph displays a line near 1		
BL9	Physical simulation	Line grows closer to 1 with vibrations		
BL10	At least (5-6) * 2hz sample rate	100 samples take at most 8.33 seconds	Pass	Same test as BM8
BL11	Resource	Uses less than 10% CPU 500MB RAM	Pass	https://youtu.be/NLnzt0uMcbo
BL12	Paint tracking	Paint tracks the same with/without logging	Pass	https://youtu.be/fnLcD8yNiMw

White box tests

Mouse Client

Test ID	Name	Expected	Outcome	Testing video reference
WM1	Config import	URL variable stores valid URL, as does password and username, given authentication	Pass	https://youtu.be/Ei-Aadkn9Qo
WM2	Menu messages	All message id's match their expected ID	Pass	https://youtu.be/Vuy3lprjttE

Training

Test ID	Name	Expected	Outcome	Testing video reference
WT1	Fourier transform	Spike at index 5, spike at indexes 7,90 and 450	Pass	https://youtu.be/cIl2Qw9EYbU
WT2	Logs	Same number of files detected, all file paths are correct	Pass	https://youtu.be/jFQIb-b1Fek
WT3	Matrices multiply	GPU product same as CPU and online	Pass	https://youtu.be/-t0q1tfzj8Q
WT4	Small network train	Weights change as expected, error decreases	Pass	https://youtu.be/u5HEzDebLs8
WT5	Fourier Transform Iterations	N log n iterations	Pass	https://youtu.be/Bb9a-JilOpE

Live detection

Test ID	Name	Expected	Outcome	Testing video reference
WL1	Graph	NN output is expected, as shown on graph	Pass	https://youtu.be/zFJVLs9UAj8
WL2	Parkinson's simulation	Normal graph is smooth, simulated graph undulates with sharp peaks		
WL3	Watch mouse refresh rate	Equal to roughly the refresh rate of the mouse in use	Pass	https://youtu.be/Sw4uYR5GSHo https://www.razer.com/gb-en/gaming-mice/razer-deathadder-essential

Evaluation

Interviewer report

The purpose of the project was to train a neural network to distinguish data simulating parkinsonian type tremor and the data set produced by healthy individuals. On reviewing the report, I was very impressed by the level of complexity used and the accuracy of the detection achieved. The particular use of a GPU accelerated matrix library to speed up network training exceeded the original brief and demonstrated a high level of competency, in carrying out this A level project.

The two interactive components William showed to me for data collection and detection appeared to fulfil the brief but I do think it could be difficult to understand the graph so would have liked another option to show a more user-friendly looking result such as a simple percentage or a bar scale. Also I would have liked to see more techniques for simulation since I think there should be more than the two highlighted in the investigation. I am satisfied with how personal data is handled in the project and think it's a worthy addition that data can be collected from any computer with an internet connection.

As a non-specialist in this area of computer science I believe the project achieved its stated intensions and showed an impressive 93% accuracy compared to the 75% hoped for. It should be emphasised that this is a computer science project using simulated data sets rather than a project intended for clinical application. I cannot comment in detail on the technical details of the training program but the results clearly meet the purpose. Overall, I think the investigation progressed well to fully meet its goals.

Jonathan Marshall

Evaluation of interview

I believe the interview was very positive and that Johnathon was pleased with the progress the project has made from its inception. I'm glad to hear he thinks it is of a high A-level standard but would like to focus on the improvements highlighted.

I agree with the main criticism, that the simulation lacked some detail and didn't represent the full symptomology of Parkinson's, I would like to, in future add a user interface to simulate a variety of aspects of Parkinson's through a GUI, not dissimilar to that of a volume mixer, granted further research in the symptomology allowing the inclusion of more, more complex parkinsonian characteristics. I do believe though, that in the solutions current state, there is already sufficient complexity especially given the time constraint such research would have only tightened further.

I believe the comment on the graphs usability to be valid and think this is an area that should be improved given its ease and benefit to the solution as a whole, especially given it should be useable by the widest audience possible.

Overall, I am happy with the comments made and believe them to be valid for the investigation. Given time I would like to add the components discussed.

Analysis of requirements

1. A system that can decide with a certainty of at least 75% whether a person is believed to have PD or not

   This is the key requirement, that the solution can correctly classify samples with at least 75% accuracy. I am very pleased that the solution has surpassed this value being able to correctly classify samples with an accuracy of 92.84% (As demonstrated by test BT13). I think this could be easily transferred to real life parkinsonian data and would hope to, in future be able to collect large amounts of said data using the mouse client and train the network further on such data. Despite my misfortune of not having any opportunity to collect Parkinsonian data I strongly believe that such a large network accompanied with optimised training procedures would be able to show better than 75% accuracy, as it has demonstrated a clear ability to classify the basic symptoms from real mouse data with superposed, randomised symptoms.

2. Collect highest possible detail mouse data
   a. Is capable of clearly showing tremors
      i. Mouse data is collected at the maximum resolution of the mouse which is stored

   The mouse client has shown itself to be highly optimised and capable of streaming mouse data at the greatest resolution possible, without data loss or corruption, this is demonstrated by test BM7 in the testing video BM5 in which you can see the mouse data collected is clearly greater than that windows uses natively. Thus, through the use of raw input messages the mouse client reads data from the lowest level possible – straight from the mouse driver, I do not believe that this could be improved in future as it is hardware dependent, if the project were to grow, I would invest in high resolution mice for the best results.

   b. Sampled at a frequency of at least 6*2Hz
      i. Due to tremors occurring at 4-6Hz multiplied by 2 due to Nyquist's theorem

   The sample rate is shown to be far greater than 12Hz as the test BM8 clearly demonstrates. This test showed that the sample rate achievable with the given mouse was far greater than 12Hz as it was in fact 980Hz, within the margin of error of the sample rate for the given mouse listed on its specification (1000Hz), this leaves the network with far more data to work on allowing the recognition of component frequencies up to 500Hz, well above 12Hz, consequently I believe this requirement to be fully for filled as it is capable of sampling at far greater frequencies than required, for a more detailed look into the sample rate calculation, see test WL3.

      ii. Mouse refresh rate detected

   This requirement ensures that the refresh rate has been calculated from the given mouse data, this is required for the output of the Fourier transform to be scaled to the correct frequencies without it the crucial recognition of 4-6Hz components would not be possible. Consequently, the trainer and live detection members of the project must be able to calculate this frequency with a high degree of accuracy, since its directly proportional to the interval the solution used the interval instead and was able to calculate it to be 0.835ms giving an accuracy of 81% given the mouse used sampled at 1000Hz giving a sampling interval of 1 millisecond.

c. Recognise new mouse devices when they are connected
    i. Record available mouse information

This requirement prevents the solution missing any potential mouse data or not logging any at all (in the case of virtual mice sometimes used for RGB devices). It has been met by the solution as it automatically logs data from all available devices when they are moved and a new message is posted. This automatic device switching is shown in test BM9.

d. Does not crash when all input devices are disconnected

This requirement prevents the program from crashing due to a device being disconnected and then logging halting, potentially for a long period losing a lot of potentially useful data. The solution has proven to not be affected by the disconnection of all input devices in the test BM10.

3. Data collected is stored in a clear and known format that easy to view while taking minimal space

This has been achieved through my log to CSV option in the mouse client context menu which searches for all mouse logs stored in the log directory and exports their contents to a CSV file in plain text, this file can be viewed in Microsoft excel and the majority of other spreadsheeting applications. This tool is demonstrated in test BM5.

a. Data can be exported to Excel for visualization
    i. Exported data is split into three columns, mouse X,Y coordinates and time

The data has been divided accordingly in the CSV file exported into the corresponding fields with each sample being stored as a single record, the layout is demonstrated in test BM5.

    ii. All logs have headers that are easy to understand

Each log has been labelled with a plain text header denoting the source of the samples, this allows the user easy reference to the data, should the chose to delete it. The titling can be seen in test BM5 as well.

b. Data is stored in binary for minimal file size
    i. A predefined data length is established that allows for storage at maximum resolution across all input devices

This data length has been documented in my design and technical solution, it can be seen when I inspect the contents of a binary sample log and is a 4-byte integer for the X delta and Y delta and a 8 byte integer for the times delta, in micro-seconds. The inspection of the binary file can be seen in test BM11

c. Data can be continually stored with a stream

The file stream to a binary file with extension ".dopbf" (detection of Parkinson's binary file) is well documented in my design, the implementation of this stream is demonstrated in test BM1.

    i. Stream does not pause

The stream is shown in test BM1 and there are a variety of tests following being: BM9, BM10 and BM11 that test the robustness of the sample logger

    ii. Stream does not require large memory buffers >500MB

The memory buffer used in the solution to store samples until upload was kept to a minimum meaning the entire solution to log samples required only 2MB!

4. Data is transmitted to a web server for storage

Data recorded is uploaded to an FTP server specified in the configuration file, tests for the configuration file are BM3 and WM1, and for upload are BM6 and BM13.

a. Data being transmitted is encrypted
    i. Can be quickly encrypted and decrypted efficiently without utilizing to many system resources

This is handled within the .NET frameworks FTP client which automatically applies SSL when the chosen server supports it, adding a level of security sufficient for this project.

b. No data is lost as it is stored

This is demonstrated in test BM13, which comparted the contents of the FTP upload and the local files, it proved that no data was lost or corrupted from a file upload.

c. Data can be accessed at all times

This is a property of the FTP server in use as they can be concurrently accessed by a number of users and can have access permissions set denying the general public from reading the data, allowing only authorised admins.

d. Restarts do not cause the loss of any data, i.e. the data must be streamed to the sever

This is a property of buffered uploads as often as possible, although some data may be lost, it is minimal and any system will have data losses in the event of unexpected halting, consequently I believe this requirement to be met.

5. Create a model of Parkinson like mouse movement

   This has been achieved by the super-position of a sign wave and "cog wheel motion" on known healthy mouse data. The change is shown graphically in my design and is demonstrated in the testing in test WL2.

   a. Model is scaled the same as mouse input

   This is done through the normalisation of the mouse data to have a mean of 0 and variance 1 and is shown in test WL2

   b. Model uses a smoothed "healthy" input

   Likewise, this has been achieved by normalising the data, which can be seen in test WL2

   c. Has a 4-6Hz sine wave applied to it

   This is achieved through the add sine wave function, documented in test WL2

   d. Contains noise with sporadic peaks

   Achieved with the "Add cog wheel" method documented again in test WL2

6. FFT used to Pre-process

   a. Uses the Cooley Tukey recursive algorithm

   The algorithm and its outputs are documented in test WL1 in which the Fourier transform algorithm employed was ran on a number of time domain inputs and successfully decomposed them to their constituent frequency domain interpretation.

   b. Can be ran in n*log(n) time

   This is clearly demonstrated in test WT5 in which the computational complexity was analysed and shown to be clearly proportional to the function n*log(n)

7. Multi-Layer recurrent neural net training

   a. Model can be trained with minimal compute power, taking a matter of days of compute time on a i7 machine

   This has been achieved with the combination of a CPU and GPU library, allowing the trainer to be easily recompiled for operation on either. The GPU library allows for the large network used to be trained in a reasonable time frame. BT8, BT12 and WT4.

8. Multi-layer neural net to process user mouse data

   The dimensions of the network are 1024 inputs to 768 to 512 to 256 to 1 output, giving the network 4 hidden layers. The dimensions of the network are displayed when they model store is imported to the program which is demonstrated in BT4.

   a. Is more than 75% accurate

   The accuracy of the neural network library to classify both parkinsonian data and quickly converge to classify XOR inputs correctly is demonstrated BT8 and WT4 respectively.

   b. Displays output clearly if requested

   This is achieved during training by outputting the classification for every class imported at that time every 100 iterations, this can be seen in test BT8.

   The output is then shown graphically when the network is running within the live detection client, demonstrated in test BL5 and BL7

   c. Does not crash given any extreme mouse data

   This is handled by the normalisation of data as shown in WL2.

9. The entire solution can be ran on a client machine with minimal impact

   This is the property of the live detection and mouse client components of the project which each utilise the system tray to be accessible but have the least impact on the user,

   a. Solution uses no more than 10% of the CPU (on an 4 threaded processor)

   This is demonstrated when I measure the resource consumption of both clients, demonstrated in test BM12 and BL11 which both show a very low CPU utilisation, with the mouse client having peaks of just 5%.

   b. Solution uses less than 500MB of RAM

   This is demonstrated when I measure the resource consumption of both clients, demonstrated in test BM12 and BL11 which both show an extremely low memory usage with the mouse client using just 2MB and the live detector using 75MB of ram.

   c. Causes no mouse "stutters" or unresponsiveness

   This has been demonstrated in test BL12, which shows the live detection program having no impact on mouse tracking within programs.

Summary

Overall, I deem the investigation to have a positive outcome that, yes – a neural network can be used to classify parkinsonian like symptoms with a high degree of accuracy; that could potentially be transferred to the clinical environment, provided the mouse data was collected in a more controlled environment and that there was an equal representation of people who were suffering from Parkinson's as those who were not. I am upset that I was unable to source any patients with the symptoms but feel that despite this setback the investigation has progressed through the application of complex algorithms such as the Cooley Tukey recursive fast Fourier transform, the GPU accelerated matrix maths library used in conjunction with a neural network processing library to run and train a network of arbitrary size in the most efficient manner possible given the available hardware.

These aspects have combined to form a solution that has met all requirements listed when I began and has been able to exceed my expectations in its accuracy and stability.

Improvements

If I was to continue the investigation, I would invest time in the following subject areas:

- Implementation of an LSTM recurrent neural network to add context to the classification process, this would require a far more complex neural network library as the processing of such a network involves many more stages. Consequently, the training techniques used differ wildly to a feed forward neural network so would require further research into the subject would be needed and a deeper understanding of the calculus applied.
- Addition of further simulation techniques, the addition of further simulations would require a deeper understanding into the symptomology of Parkinson's but I believe this could be achieved by consulting an expert in that field in future.
- User interface to "mix" the intensity of such simulations, this would take form using the same graphics library used earlier in the project but would demand a far more complex user interface model as sliders would have to be updated according to mouse user input where as the current graphics library need only respond to resizing of the window

Bibliography

REPLACE SPACES WITH UNDERSCORES

Adams, W. R., November 30, 2017. *High-accuracy detection of early Parkinson's Disease using multiple characteristics of finger movement while typing.* [Online]
Available at:
https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0188226
[Accessed 14 September 2018].

Alex Graves, J. S., 2008. *Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks.* [Online]
Available at: https://papers.nips.cc/paper/3449-offline-handwriting-recognition-with-multidimensional-recurrent-neural-networks
[Accessed September 2018].

Amazon, 2018. *AWS Free Tier.* [Online]
Available at: https://aws.amazon.com/free/?nc2=h_ql_pr
[Accessed 2 October 2018].

Asteroids, 2018. *CPU performance.* [Online]
Available at: https://asteroidsathome.net/boinc/cpu_list.php
[Accessed Dec 2018].

Brownlee, J., 2016. *Linear Regression for Machine Learning.* [Online]
Available at: https://machinelearningmastery.com/linear-regression-for-machine-learning/
[Accessed September 2018].

Brownlee, J., 2016. *Naive Bayes for Machine Learning.* [Online]
Available at: https://machinelearningmastery.com/naive-bayes-for-machine-learning/
[Accessed September 2018].

Brownlee, J., 2016. *Support Vector Machines for Machine Learning.* [Online]
Available at: https://machinelearningmastery.com/support-vector-machines-for-machine-learning/
[Accessed September 2018].

Carmack, M. B. &. C., 2016. *How Computer Mice Work.* [Online]
Available at: https://computer.howstuffworks.com/mouse5.htm
[Accessed sep 2018].

Chamikara, M., 2014. *Can someone recommend what is the best percent of divided the training data and testing data in neural network 75:25 or 80:20 or 90:10 ?.* [Online]
Available at:
https://www.researchgate.net/post/can_someone_recommend_what_is_the_best_percent_of_divided_the_training_data_and_testing_data_in_neural_network_7525_or_8020_or_9010
[Accessed 3 October 2018].

Codeka, 2009. *Mouse input and multiple monitors.* [Online]
Available at: https://www.gamedev.net/forums/topic/528500-mouse-input-and-multiple-monitors/
[Accessed September 2018].

Dernoncourt, F., 2016. *A simple explanation of Naive Bayes Classification.* [Online]
Available at: https://stackoverflow.com/questions/10059594/a-simple-explanation-of-naive-bayes-classification
[Accessed September 2018].

Freeman, J., 2018. *MouseTracker*. [Online]
Available at: http://www.mousetracker.org/
https://en.wikipedia.org/wiki/Mouse_tracking
[Accessed Sep 2018].
Freeman, L., 2016. *RNN vs CNN at a high level*. [Online]
Available at: https://datascience.stackexchange.com/questions/11619/rnn-vs-cnn-at-a-high-level
[Accessed September 2018].
Fumo, D., 2017. *Types of Machine Learning Algorithms You Should Know*. [Online]
Available at: https://towardsdatascience.com/types-of-machine-learning-algorithms-you-should-know-953a08248861
[Accessed 6 September 2018].
Gandhi, R., 2018. *Introduction to Machine Learning Algorithms: Linear Regression*. [Online]
Available at: https://towardsdatascience.com/introduction-to-machine-learning-algorithms-linear-regression-14c4e325882a
[Accessed September 2018].
Gupta, P., 2017. *Decision Trees in Machine Learning*. [Online]
Available at: https://towardsdatascience.com/decision-trees-in-machine-learning-641b9c4e8052
[Accessed September 2018].
Hope, C., 2017. *How do computers connect to each other over the Internet?*. [Online]
Available at: https://www.computerhope.com/issues/ch001358.htm
[Accessed 31 September 2018].
Jankovic, J., March 14, 2008.. *Parkinson’s disease: clinical features and diagnosis*. [Online]
Available at: https://jnnp.bmj.com/content/79/4/368
[Accessed 11 September 2018].
L.Abrahamsea, M. F. D., 2016. *Sequence learning in Parkinson's disease: Focusing on action dynamics and the role of dopaminergic medication*. [Online]
Available at:
https://www.sciencedirect.com/science/article/pii/S0028393216303645
[Accessed Sep 2018].
Maladkar, K., 2018. *6 Types of Artificial Neural Networks Currently Being Used in Machine Learning*. [Online]
Available at: https://analyticsindiamag.com/6-types-of-artificial-neural-networks-currently-being-used-in-todays-technology/
[Accessed September 2018].
Microsoft, 2018. *About Raw Input*. [Online]
Available at: https://docs.microsoft.com/en-us/windows/desktop/inputdev/about-raw-input
[Accessed September 2018].
Microsoft, 2018. *Downloading and uploading files on OneDrive (REST)*. [Online]
Available at: https://docs.microsoft.com/en-us/previous-versions/office/developer/onedrive-live-sdk/dn659726(v=office.15)#upload_a_file
[Accessed 1 October 2018].
Microsoft, 2018. *Walkthrough: Creating Windows Desktop Applications (C++)*. [Online]
Available at: https://msdn.microsoft.com/en-us/library/bb384843.aspx
[Accessed 4 October 2018].
Notebaert, M. F. L. R. E. L. A. P. S. W., 2017. *The effect of dopaminergic medication on conflict adaptation in Parkinson's disease*. [Online]

Available at: https://onlinelibrary.wiley.com/doi/full/10.1111/jnp.12131
[Accessed Sep 2018].
Parkinson's UK, 2017. [Online]
Available at: https://www.parkinsons.org.uk/sites/default/files/2018-01/Prevalence%20%20Incidence%20Report%20Latest_Public_2.pdf
[Accessed Aug 2018].
Ryen W. White, P. M. D. &. E. H., 2018. *Brief Communication | OPEN | Published: 23 April 2018.* [Online]
Available at: https://www.nature.com/articles/s41746-018-0016-6
[Accessed 13 September 2018].
Sanjeevi, M., 26 September 2017. *Different types of Machine Learning and their types.*. [Online]
Available at: https://medium.com/deep-math-machine-learning-ai/different-types-of-machine-learning-and-their-types-34760b9128a2
[Accessed 5 September 2018].
Statista, 2018. *Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions).* [Online]
Available at: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/
[Accessed 1 October 2018].
statista, 2019. *Time spent per day with media content via computer in the United States from 2012 to 2018, by type (in minutes).* [Online]
Available at: https://www.statista.com/statistics/469995/time-spent-desktop-laptop-media-type-usa/
[Accessed 2 Feb 2018].
Wikapedia, 2018. *Convolutional Neural Network.* [Online]
Available at: https://en.wikipedia.org/wiki/Convolutional_neural_network
[Accessed September 2018].
Wikapedia, 2018. *FeedForward neural network.* [Online]
Available at: https://en.wikipedia.org/wiki/Feedforward_neural_network
[Accessed September 2018].
Wikipedia, 2018. *Computer mouse.* [Online]
Available at: https://en.wikipedia.org/wiki/Computer_mouse
[Accessed Sep 2018].
Wikipedia, 2018. *Cooley-Tukey FFT.* [Online]
Available at: https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm
[Accessed 4 October 2018].
Wikipedia, 2018. *Feedforward neural network.* [Online]
Available at: https://en.wikipedia.org/wiki/Feedforward_neural_network
[Accessed September 2018].
Wikipedia, 2018. *Foureier transform.* [Online]
Available at: https://en.wikipedia.org/wiki/Fourier_transform
[Accessed 4 October 2018].
Wikipedia, 2018. *Peer to Peer.* [Online]
Available at: https://en.wikipedia.org/wiki/Peer-to-peer
[Accessed 31 September 2018].